User's Guide

SIESTA 4.0.2

July 19, 2018

Emilio Artacho	*CIC-Nanogune and University of Cambridge*
José María Cela	*Barcelona Supercomputing Center*
Julian D. Gale	*Curtin University of Technology, Perth*
Alberto García	*Institut de Ciència de Materials, CSIC, Barcelona*
Javier Junquera	*Universidad de Cantabria, Santander*
Richard M. Martin	*University of Illinois at Urbana-Champaign*
Pablo Ordejón	*Centre de Investigació en Nanociència i Nanotecnologia, (CSIC-ICN), Barcelona*
Daniel Sánchez-Portal	*Unidad de Física de Materiales, Centro Mixto CSIC-UPV/EHU, San Sebastián*
José M. Soler	*Universidad Autónoma de Madrid*

`http://www.uam.es/siesta`

Contents

1 INTRODUCTION

This Reference Manual contains descriptions of all the input, output and execution features of SIESTA*, but is not really a tutorial introduction to the program. Interested users can find tutorial material prepared for* SIESTA *schools and workshops at the project's web page* `http://www.uam.es/siesta`.

SIESTA (Spanish Initiative for Electronic Simulations with Thousands of Atoms) is both a method and its computer program implementation, to perform electronic structure calculations and *ab initio* molecular dynamics simulations of molecules and solids. Its main characteristics are:

- It uses the standard Kohn-Sham selfconsistent density functional method in the local density (LDA-LSD) and generalized gradient (GGA) approximations, as well as in a non local functional that includes van der Waals interactions (VDW-DF).
- It uses norm-conserving pseudopotentials in their fully nonlocal (Kleinman-Bylander) form.
- It uses atomic orbitals as a basis set, allowing unlimited multiple-zeta and angular momenta, polarization and off-site orbitals. The radial shape of every orbital is numerical and any shape can be used and provided by the user, with the only condition that it has to be of finite support, i.e., it has to be strictly zero beyond a user-provided distance from the corresponding nucleus. Finite-support basis sets are the key for calculating the Hamiltonian and overlap matrices in $O(N)$ operations.
- Projects the electron wavefunctions and density onto a real-space grid in order to calculate the Hartree and exchange-correlation potentials and their matrix elements.
- Besides the standard Rayleigh-Ritz eigenstate method, it allows the use of localized linear combinations of the occupied orbitals (valence-bond or Wannier-like functions), making the computer time and memory scale linearly with the number of atoms. Simulations with several hundred atoms are feasible with modest workstations.
- It is written in Fortran 95 and memory is allocated dynamically.
- It may be compiled for serial or parallel execution (under MPI).

It routinely provides:

- Total and partial energies.
- Atomic forces.
- Stress tensor.
- Electric dipole moment.
- Atomic, orbital and bond populations (Mulliken).
- Electron density.

And also (though not all options are compatible):

- Geometry relaxation, fixed or variable cell.
- Constant-temperature molecular dynamics (Nose thermostat).
- Variable cell dynamics (Parrinello-Rahman).
- Spin polarized calculations (collinear or not).
- k-sampling of the Brillouin zone.
- Local and orbital-projected density of states.
- COOP and COHP curves for chemical bonding analysis.
- Dielectric polarization.
- Vibrations (phonons).
- Band structure.
- Ballistic electron transport (through TranSiesta)

Starting from version 3.0, Siesta includes the TranSiesta module. TranSiesta provides the ability to model open-boundary systems where ballistic electron transport is taking place. Using TranSiesta one can compute electronic transport properties, such as the zero bias conductance and the I-V characteristic, of a nanoscale system in contact with two electrodes at different electrochemical potentials. The method is based on using non equilibrium Green's functions (NEGF), that are constructed using the density functional theory Hamiltonian obtained from a given electron density. A new density is computed using the NEGF formalism, which closes the DFT-NEGF self consistent cycle.

For more details on the formalism, see the main TranSiesta reference cited below. A section has been added to this User's Guide, that describes the necessary steps involved in doing transport calculations, together with the currently implemented input options.

References:

- "Unconstrained minimization approach for electronic computations that scales linearly with system size" P. Ordejón, D. A. Drabold, M. P. Grumbach and R. M. Martin, Phys. Rev. B **48**, 14646 (1993); "Linear system-size methods for electronic-structure calculations" Phys. Rev. B **51** 1456 (1995), and references therein.

  Description of the order-N eigensolvers implemented in this code.

- "Self-consistent order-N density-functional calculations for very large systems" P. Ordejón, E. Artacho and J. M. Soler, Phys. Rev. B **53**, 10441, (1996).

  Description of a previous version of this methodology.

- "Density functional method for very large systems with LCAO basis sets" D. Sánchez-Portal, P. Ordejón, E. Artacho and J. M. Soler, Int. J. Quantum Chem., **65**, 453 (1997).

  Description of the present method and code.

- "Linear-scaling ab-initio calculations for large and complex systems" E. Artacho, D. Sánchez-Portal, P. Ordejón, A. García and J. M. Soler, Phys. Stat. Sol. (b) **215**, 809 (1999).

  Description of the numerical atomic orbitals (NAOs) most commonly used in the code, and brief review of applications as of March 1999.

- "Numerical atomic orbitals for linear-scaling calculations" J. Junquera, O. Paz, D. Sánchez-Portal, and E. Artacho, Phys. Rev. B **64**, 235111, (2001).

  Improved, soft-confined NAOs.

- "The SIESTA method for ab initio order-N materials simulation" J. M. Soler, E. Artacho, J.D. Gale, A. García, J. Junquera, P. Ordejón, and D. Sánchez-Portal, J. Phys.: Condens. Matter **14**, 2745-2779 (2002)

  Extensive description of the SIESTA method.

- "Computing the properties of materials from first principles with SIESTA", D. Sánchez-Portal, P. Ordejón, and E. Canadell, Structure and Bonding **113**, 103-170 (2004).

  Extensive review of applications as of summer 2003.

- "Density-functional method for nonequilibrium electron transport", Mads Brandbyge, Jose-Luis Mozos, Pablo Ordejón, Jeremy Taylor, and Kurt Stokbro, Phys. Rev. B **65**, 165401 (2002).

  Description of the TRANSIESTA method.

For more information you can visit the web page `http://www.uam.es/siesta`.

2 COMPILATION

2.1 The building directory

Rather than using the top-level Src directory as building directory, the user has to use an ad-hoc building directory (by default the top-level `Obj` directory, but it can be any (new) directory in the top level). The building directory will hold the object files, module files, and libraries resulting from the compilation of the sources in `Src`. The `VPATH` mechanism of modern `make` programs is used. This scheme has many advantages. Among them:

- The Src directory is kept pristine.
- Many different object directories can be used concurrently to compile the program with different compilers or optimization levels.

If you just want to compile the program, go to `Obj` and issue the command:

```
sh ../Src/obj_setup.sh
```

to populate this directory with the minimal scaffolding of makefiles, and then make sure that you create or generate an appropriate `arch.make` file (see below, in Sect. 2.2). Then, type

```
make
```

The executable should work for any job. (This is not exactly true, since some of the parameters in the atomic routines are still hardwired (see `Src/atmparams.f`), but those would seldom need to be changed.)

To compile utility programs (those living in `Util`), you can just simply use the provided makefiles, typing "make" as appropriate.

2.1.1 Multiple-target compilation

The mechanism described here can be repeated in other directories at the same level as Obj, with different names. In this way one can compile as many different versions of the SIESTA executable as needed (for example, with different levels of optimization, serial, parallel, debug, etc), by working in separate building directories.

Simply provide the appropriate arch.make, and issue the setup command above. To compile utility programs, you need to use the form:

```
make OBJDIR=ObjName
```

where `ObjName` is the name of the object directory of your choice. Be sure to type `make clean` before attempting to re-compile a utility program.

(The pristine Src directory should be kept "clean", without objects, or else the compilation in the build directories will get confused)

2.2 The arch.make file

The compilation of the program is done using a `Makefile` that is provided with the code. This `Makefile` will generate the executable for any of several architectures, with a minimum of tuning required from the user and encapsulated in a separate file called `arch.make`.

You are strongly encouraged to look at `Src/Sys/DOCUMENTED-TEMPLATE.make` for information about the fine points of the `arch.make` file. You can also get inspiration by looking at the actual `arch.make` examples in the `Src/Sys` subdirectory. If you intend to create a parallel version of SIESTA, make sure you have all the extra support libraries (`MPI, scalapack, blacs...` (see Sect. 16).

Optionally, the command `../Src/configure` will start an automatic scan of your system and try to build an `arch.make` for you. Please note that the configure script might need some help in

order to find your Fortran compiler, and that the created `arch.make` may not be optimal, mostly in regard to compiler switches and preprocessor definitions, but the process should provide a reasonable first version. Type `../Src/configure --help` to see the flags understood by the script, and take a look at the `Src/Confs` subdirectory for some examples of their explicit use.

3 EXECUTION OF THE PROGRAM

A fast way to test your installation of SIESTA and get a feeling for the workings of the program is implemented in directory `Tests`. In it you can find several subdirectories with pre-packaged FDF files and pseudopotential references. Everything is automated: after compiling SIESTA you can just go into any subdirectory and type `make`. The program does its work in subdirectory `work`, and there you can find all the resulting files. For convenience, the output file is copied to the parent directory. A collection of reference output files can be found in `Tests/Reference`. Please note that small numerical and formatting differences are to be expected, depending on the compiler. (For non-standard execution environments, including queuing systems, have a look at the Scripts in Tests/Scripts, and see also Sect. 16.)

Other examples are provided in the `Examples` directory. This directory contains basically `.fdf` files and the appropriate pseudopotential generation input files. Since at some point you will have to generate your own pseudopotentials and run your own jobs, we describe here the whole process by means of the simple example of the water-molecule. It is advisable to create independent directories for each job, so that everything is clean and neat, and out of the `siesta` directory, so that one can easily update version by replacing the whole `siesta` tree. Go to your favorite working directory and:

```
$ mkdir h2o
```

```
$ cd h2o
```

```
$ cp path-to-package/Examples/H2O/h2o.fdf .
```

You need to make the siesta executable visible in your path. You can do it in many ways, but a simple one is

```
ln -s path-to-package/Obj/siesta .
```

We need to generate the required pseudopotentials. (We are going to streamline this process for this time, but you must realize that this is a tricky business that you must master before using SIESTA responsibly. Every pseudopotential must be thoroughly checked before use. Please refer to the ATOM program manual for details regarding what follows.)

NOTE: The ATOM program is no longer bundled with SIESTA, but academic users can dowload it from the SIESTA webpage at `www.icmab.es/siesta`.

```
$ cd path/to/atom/package/
```

(Compile the program following the instructions)

```
$ cd Tutorial/PS_Generation/O
```

```
$ cat O.tm2.inp
```

This is the input file, for the oxygen pseudopotential, that we have prepared for you. It is in a

standard (but ancient and obscure) format that you will need to understand in the future:

```
------------------------------------------------------------
   pg      Oxygen
        tm2  2.0
 n=0  c=ca
       0.0       0.0       0.0       0.0       0.0       0.0
    1    4
    2    0     2.00      0.00
    2    1     4.00      0.00
    3    2     0.00      0.00
    4    3     0.00      0.00
   1.15     1.15     1.15     1.15
------------------------------------------------------------
```

To generate the pseudopotential do the following;

```
$ sh ../../Utils/pg.sh O.tm2.inp
```

Now there should be a new subdirectory called O.tm2 (O for oxygen) and `O.tm2.vps` (binary) and `O.tm2.psf` (ASCII) files.

```
$ cp O.tm2.psf path-to-working-dir/h2o/O.psf
```

copies the generated pseudopotential file to your working directory. (The unformatted and ASCII files are functionally equivalent, but the latter is more transportable and easier to look at, if you so desire.) The same could be repeated for the pseudopotential for H, but you may as well copy `H.psf` from `Examples/Vps/` to your `h2o` working directory.

Now you are ready to run the program:

```
./siesta < h2o.fdf | tee h2o.out
```

(If you are running the parallel version you should use some other invocation, such as `mpirun -np 2 siesta ...`, but we cannot go into that here — see Sect. 16).

After a successful run of the program, you should have several files in your directory including the following:

- fdf.log (contains all the data used, explicit or chosen by default)
- O.ion and H.ion (complete information about the basis and KB projectors)
- h2o.XV (contains positions and velocities)
- h2o.STRUCT_OUT (contains the final cell vectors and positions in "crystallographic" format)
- h2o.DM (contains the density matrix to allow a restart)
- h2o.ANI (contains the coordinates of every MD step, in this case only one)
- h2o.FA (contains the forces on the atoms)

- h2o.EIG (contains the eigenvalues of the Kohn-Sham Hamiltonian)
- h2o.xml (XML marked-up output)

The prefix h2o of all these files is the `SystemLabel` specified in the input h2o.fdf file (see FDF section below). The standard output of the program, that you have already seen passing on the screen, was copied to file h2o.out by the tee command. Have a look at it and refer to the output-explanation section if necessary. You may also want to look at the fdf.log file to see all the default values that siesta has chosen for you, before studying the input-explanation section and start changing them.

Now look at the other data files in `Examples` (all with an .fdf suffix) choose one and repeat the process for it.

4 THE FLEXIBLE DATA FORMAT (FDF)

The main input file, which is read as the standard input (unit 5), contains all the physical data of the system and the parameters of the simulation to be performed. This file is written in a special format called FDF, developed by Alberto García and José M. Soler. This format allows data to be given in any order, or to be omitted in favor of default values. Refer to documentation in ∼/siesta/Src/fdf for details. Here we offer a glimpse of it through the following rules:

- The FDF syntax is a 'data label' followed by its value. Values that are not specified in the datafile are assigned a default value.
- FDF labels are case insensitive, and characters - _ . in a data label are ignored. Thus, LatticeConstant and lattice_constant represent the same label.
- All text following the # character is taken as comment.
- Logical values can be specified as T, true, .true., yes, F, false, .false., no. Blank is also equivalent to true.
- Character strings should **not** be in apostrophes.
- Real values which represent a physical magnitude must be followed by its units. Look at function fdf_convfac in file ∼/siesta/Src/fdf/fdf.f for the units that are currently supported. It is important to include a decimal point in a real number to distinguish it from an integer, in order to prevent ambiguities when mixing the types on the same input line.
- Complex data structures are called blocks and are placed between '%block label' and a '%endblock label' (without the quotes).
- You may 'include' other FDF files and redirect the search for a particular data label to another file. If a data label appears more than once, its first appearance is used.
- If the same label is specified twice, the first one takes precedence.

- If a label is misspelled it will not be recognized (there is no internal list of "accepted" tags in the program). You can check the actual value used by siesta by looking for the label in the output *fdf.log* file.

These are some examples:

```
SystemName      Water molecule  # This is a comment
SystemLabel     h2o
SpinPolarized        yes
SaveRho
NumberOfAtoms         64
LatticeConstant       5.42 Ang
%block LatticeVectors
         1.000  0.000  0.000
         0.000  1.000  0.000
         0.000  0.000  1.000
%endblock LatticeVectors
KgridCutoff < BZ_sampling.fdf

# Reading the coordinates from a file
%block AtomicCoordinatesAndAtomicSpecies < coordinates.data

# Even reading more FDF information from somewhere else
%include mydefaults.fdf
```

The file *fdf.log* contains all the parameters used by SIESTA in a given run, both those specified in the input fdf file and those taken by default. They are written in fdf format, so that you may reuse them as input directly. Input data blocks are copied to the fdf.log file only if you specify the *dump* option for them.

5 PROGRAM OUTPUT

5.1 Standard output

SIESTA writes a log of its workings to standard output (unit 6), which is usually redirected to an "output file".

A brief description follows. See the example cases in the siesta/Tests directory for illustration.

The program starts writing the version of the code which is used. Then, the input FDF file is dumped into the output file as is (except for empty lines). The program does part of the reading and digesting of the data at the beginning within the `redata` subroutine. It prints some of the information it digests. It is important to note that it is only part of it, some other information being accessed by the different subroutines when they need it during the run (in the spirit of FDF input). A complete list of the input used by the code can be found at the end in the file `fdf.log`, including defaults used by the code in the run.

After that, the program reads the pseudopotentials, factorizes them into Kleinman-Bylander form, and generates (or reads) the atomic basis set to be used in the simulation. These stages are documented in the output file.

The simulation begins after that, the output showing information of the MD (or CG) steps and the SCF cycles within. Basic descriptions of the process and results are presented. The user has the option to customize it, however, by defining different options that control the printing of informations like coordinates, forces, $\vec{k}$ points, etc. The options are discussed in the appropriate sections, but take into account the behavior of the legacy **LongOutput** option, as in the current implementation might silently activate output to the main .out file at the expense of auxiliary files.

LongOutput (*logical*): SIESTA can write to standard output different data sets depending on the values for output options described below. By default SIESTA will not write most of them. They can be large for large systems (coordinates, eigenvalues, forces, etc.) and, if written to standard output, they accumulate for all the steps of the dynamics. SIESTA writes the information in other files (see Output Files) in addition to the standard output, and these can be cumulative or not.

Setting **LongOutput** to `.true.` changes the default of some options, obtaining more information in the output (verbose). In particular, it redefines the defaults for the following:

- **WriteKpoints**
- **WriteKbands**
- **WriteCoorStep**
- **WriteForces**
- **WriteEigenvalues**
- **WriteWaveFunctions**
- **WriteMullikenPop** (it sets it to 1)

The specific changing of any of these options overrides the **LongOutput** setting for it.

Default value: `.false.`

5.2 Output to dedicated files

SIESTA can produce a wealth of information in dedicated files, with specific formats, that can be used for further analysis. See the appropriate sections, and the appendix on file formats. Please take into account the behavior of **LongOutput**, as in the current implementation might silently activate output to the main .out file at the expense of auxiliary files.

6 DETAILED DESCRIPTION OF PROGRAM OPTIONS

Here follows a description of the variables that you can define in your SIESTA input file, with their data types and default values. For historical reasons the names of the tags do not have an uniform structure, and can be confusing at times.

Almost all of the tags are optional: SIESTA will assign a default if a given tag is not found when needed (see `fdf.log`).

6.1 General system descriptors

SystemName (*string*): A string of one or several words containing a descriptive name of the system (max. 150 characters).

Default value: blank

SystemLabel (*string*): A **single** word (max. 20 characters **without blanks**) containing a nickname of the system, used to name output files.

Default value: `siesta`

NumberOfSpecies (*integer*): Number of different atomic species in the simulation. Atoms of the same species, but with a different pseudopotential or basis set are counted as different species.

Default value: There is no default. You must supply this variable.

NumberOfAtoms (*integer*): Number of atoms in the simulation.

Default value: There is no default. You must supply this variable.

ChemicalSpeciesLabel (*data block*): It specifies the different chemical species that are present, assigning them a number for further identification. SIESTA recognizes the different atoms by the given atomic number.

```
%block Chemical_Species_label
   1   6   C
   2  14   Si
   3  14   Si_surface
%endblock Chemical_Species_label
```

The first number in a line is the species number, it is followed by the atomic number, and then by the desired label. This label will be used to identify corresponding files, namely, pseudopotential file, user basis file, basis output file, and local pseudopotential output file.

This construction allows you to have atoms of the same species but with different basis or pseudopotential, for example.

Negative atomic numbers are used for *ghost* atoms (see **PAO.basis**).

Atomic numbers over 200 are used to represent *synthetic atoms* (created for example as a "mixture" of two real ones for a "virtual crystal" (VCA) calculation). In this special case a new 'SyntheticAtoms' block must be present to give SIESTA information about the "ground state" of the synthetic atom.

```
%block Chemical_Species_label
   1   201 ON-0.50000
%endblock Chemical_Species_label
%block SyntheticAtoms
1               # Species index
2 2 3 4         # n numbers for valence states  with l=0,1,2,3
2.0 3.5 0.0 0.0 # occupations of valence states with l=0,1,2,3
%endblock SyntheticAtoms
```

Pseudopotentials for synthetic atoms can be created using the `mixps` and `fractional` programs in the `Util/VCA` directory.

Atomic numbers below -200 represent *ghost synthetic atoms*.

Use: This block is mandatory.

Default: There is no default. You must supply this block.

AtomicMass (*data block*): It allows the user to introduce the atomic masses of the different species used in the calculation, useful for the dynamics with isotopes, for example. If a species index is not found within the block, the natural mass for the corresponding atomic number is assumed. If the block is absent all masses are the natural ones. One line per species with the species index (integer) and the desired mass (real). The order is not important. If there is no integer and/or no real numbers within the line, the line is disregarded.

```
%block AtomicMass
   3  21.5
   1  3.2
%endblock AtomicMass
```

Default: (Block absent or empty) Natural masses assumed. For *ghost* atoms (i.e. floating orbitals), a default of 1.d30 a.u. is assigned.

6.2 Pseudopotentials

SIESTA uses pseudopotentials to represent the electron-ion interaction (as do most plane-wave codes and in contrast to so-called "all-electron" programs). In particular, the pseudopotentials are of the "norm-conserving" kind, and can be generated by the ATOM program, (see `Pseudo/README.ATOM`). Remember that **all pseudopotentials should be thoroughly tested** before using them. We refer you to the standard literature on pseudopotentials and to the ATOM manual for more information. A number of other codes (such as `APE`) can generate pseudopotentials that SIESTA can use directly (typically in the `.psf` format).

The pseudopotentials will be read by SIESTA from different files, one for each defined species (species defined either in block **ChemicalSpeciesLabel**). The name of the files should be:

Chemical_label.`vps` (unformatted) or *Chemical_label*.`psf` (ASCII)

where *Chemical_label* corresponds to the label defined in the **ChemicalSpeciesLabel** block.

This version of Siesta can handle pseudopotential files in a *deprecated* XML format designed by Junquera, Garcia, and Verstraete to enable interoperability with some versions of the ABINIT code. The directory `Util/pseudo-xml` contains a program to translate the pseudo XML files to the `.psf` form. This XML format will be soon superseded by the new PSML specification, which will be supported by forthcoming versions of Siesta and ABINIT.

6.3 Basis set and KB projectors

6.3.1 Overview of atomic-orbital bases implemented in SIESTA

The main advantage of atomic orbitals is their efficiency (fewer orbitals needed per electron for similar precision) and their main disadvantage is the lack of systematics for optimal convergence, an issue that quantum chemists have been working on for many years. They have also clearly shown that there is no limitation on precision intrinsic to LCAO. This section provides some information about how basis sets can be generated for SIESTA.

It is important to stress at this point that neither the SIESTA method nor the program are bound to the use of any particular kind of atomic orbitals. The user can feed into SIESTA the atomic basis set he/she choses by means of radial tables (see **User.Basis** below), the only limitations being: (*i*) the functions have to be atomic-like (radial functions mutiplied by spherical harmonics), and (*ii*) they have to be of finite support, i.e., each orbital becomes strictly zero beyond some cutoff radius chosen by the user.

Most users, however, do not have their own basis sets. For these users we have devised some schemes to generate basis sets within the program with a minimum input from the user. If nothing is specified in the input file, Siesta generates a default basis set of a reasonable quality that might constitute a good starting point. Of course, depending on the accuracy required in the particular problem, the user has the degree of freedom to tune several parameters that can be important for quality and efficiency. A description of these basis sets and some performance tests can be found in the references quoted below.

"Numerical atomic orbitals for linear-scaling calculations", J. Junquera, O. Paz, D. Sánchez-Portal, and E. Artacho, Phys. Rev. B **64**, 235111, (2001)

An important point here is that the basis set selection is a variational problem and, therefore, minimizing the energy with respect to any parameters defining the basis is an "ab initio" way to define them.

We have also devised a quite simple and systematic way of generating basis sets based on specifying only one main parameter (the energy shift) besides the basis size. It does not offer the best NAO results one can get for a given basis size but it has the important advantages mentioned above. More about it in:

"Linear-scaling ab-initio calculations for large and complex systems", E. Artacho, D. Sánchez-Portal, P. Ordejón, A. García and J. M. Soler, Phys. Stat. Sol. (b) **215**, 809 (1999).

In addition to SIESTA we provide the program GEN-BASIS, which reads SIESTA's input and

generates basis files for later use. GEN-BASIS can be found in `Util/Gen-basis`. It should be run from the `Tutorials/Bases` directory, using the `gen-basis.sh` script. It is limited to a single species.

Of course, as it happens for the pseudopotential, it is the responsibility of the user to check that the physical results obtained are converged with respect to the basis set used before starting any production run.

In the following we give some clues on the basics of the basis sets that SIESTA generates. The starting point is always the solution of Kohn-Sham's Hamiltonian for the isolated pseudo-atoms, solved in a radial grid, with the same approximations as for the solid or molecule (the same exchange-correlation functional and pseudopotential), plus some way of confinement (see below). We describe in the following three main features of a basis set of atomic orbitals: size, range, and radial shape.

Size: number of orbitals per atom

Following the nomenclature of Quantum Chemistry, we establish a hierarchy of basis sets, from single-ζ to multiple-ζ with polarization and diffuse orbitals, covering from quick calculations of low quality to high precision, as high as the finest obtained in Quantum Chemistry. A single-ζ (also called minimal) basis set (SZ in the following) has one single radial function per angular momentum channel, and only for those angular momenta with substantial electronic population in the valence of the free atom. It offers quick calculations and some insight on qualitative trends in the chemical bonding and other properties. It remains too rigid, however, for more quantitative calculations requiring both radial and angular flexibilization.

Starting by the radial flexibilization of SZ, a better basis is obtained by adding a second function per channel: double-ζ (DZ). In Quantum Chemistry, the *split valence* scheme is widely used: starting from the expansion in Gaussians of one atomic orbital, the most contracted Gaussians are used to define the first orbital of the double-ζ and the most extended ones for the second. For strictly localized functions there was a first proposal of using the excited states of the confined atoms, but it would work only for tight confinement (see **PAO.BasisType** `nodes` below). This construction was proposed and tested in D. Sánchez-Portal *et al.*, J. Phys.: Condens. Matter **8**, 3859-3880 (1996).

We found that the basis set convergence is slow, requiring high levels of multiple-ζ to achieve what other schemes do at the double-ζ level. This scheme is related with the basis sets used in the OpenMX project [see T. Ozaki, Phys. Rev. B **67**, 155108 (2003); T. Ozaki and H. Kino, Phys. Rev. B **69**, 195113 (2004)].

We then proposed an extension of the split valence idea of Quantum Chemistry to strictly localized NAO which has become the standard and has been used quite successfully in many systems (see **PAO.BasisType** `split` below). It is based on the idea of suplementing the first ζ with, instead of a gaussian, a numerical orbital that reproduces the tail of the original PAO outside a matching radius r_m, and continues smoothly towards the origin as $r^l(a - br^2)$, with a and b ensuring continuity and differentiability at r_m. Within exactly the same Hilbert space, the second orbital can be chosen to be the difference between the smooth one and the original PAO, which gives a basis orbital strictly confined within the matching radius r_m (smaller than the original PAO!) continuously differentiable throughout.

Extra parameters have thus appeared: one r_m per orbital to be doubled. The user can again

introduce them by hand (see **PAO.Basis** below). Alternatively, all the r_m's can be defined at once by specifying the value of the tail of the original PAO beyond r_m, the so-called split norm. Variational optimization of this split norm performed on different systems shows a very general and stable performance for values around 15% (except for the $\sim 50\%$ for hydrogen). It generalizes to multiple-ζ trivially by adding an additional matching radius per new zeta.

Note: What is actually used is the norm of the tail *plus* the norm of the parabola-like inner function.

Angular flexibility is obtained by adding shells of higher angular momentum. Ways to generate these so-called polarization orbitals have been described in the literature for Gaussians. For NAOs there are two ways for SIESTA and GEN-BASIS to generate them: (*i*) Use atomic PAO's of higher angular momentum with suitable confinement, and (*ii*) solve the pseudoatom in the presence of an electric field and obtain the $l+1$ orbitals from the perturbation of the l orbitals by the field.

So-called diffuse orbitals, that might be important in the description of open systems such as surfaces, can be simply added by specifying extra "n" shells. [See S. Garcia-Gil, A. Garcia, N. Lorente, P. Ordejon, Phys. Rev. B **79**, 075441 (2009)]

Finally, the method allows the inclusion of off-site (ghost) orbitals (not centered around any specific atom), useful for example in the calculation of the counterpoise correction for basis-set superposition errors. Bessel functions for any radius and any excitation level can also be added anywhere to the basis set.

Range: cutoff radii of orbitals.

Strictly localized orbitals (zero beyond a cutoff radius) are used in order to obtain sparse Hamiltonian and overlap matrices for linear scaling. One cutoff radius per angular momentum channel has to be given for each species.

A balanced and systematic starting point for defining all the different radii is achieved by giving one single parameter, the energy shift, i.e., the energy increase experienced by the orbital when confined. Allowing for system and physical-quantity variablity, as a rule of thumb $\Delta E_{\mathrm{PAO}} \approx 100$ meV gives typical precisions within the accuracy of current GGA functionals. The user can, nevertheless, change the cutoff radii at will.

Shape

Within the pseudopotential framework it is important to keep the consistency between the pseudopotential and the form of the pseudoatomic orbitals in the core region. The shape of the orbitals at larger radii depends on the cutoff radius (see above) and on the way the localization is enforced.

The first proposal (and quite a standard among SIESTA users) uses an infinite square-well potential. It was originally proposed and has been widely and successfully used by Otto Sankey and collaborators, for minimal bases within the ab initio tight-binding scheme, using the FIREBALL program, but also for more flexible bases using the methodology of SIESTA. This scheme has the disadavantage, however, of generating orbitals with a discontinuous derivative at r_c. This discontinuity is more pronounced for smaller r_c's and tends to disappear for long enough values of this cutoff. It does remain, however, appreciable for sensible values of r_c for those orbitals that would be very wide in the free atom. It is surprising how small an effect such a kink produces

in the total energy of condensed systems. It is, on the other hand, a problem for forces and stresses, especially if they are calculated using a (coarse) finite three-dimensional grid.

Another problem of this scheme is related to its defining the basis starting from the free atoms. Free atoms can present extremely extended orbitals, their extension being, besides problematic, of no practical use for the calculation in condensed systems: the electrons far away from the atom can be described by the basis functions of other atoms.

A traditional scheme to deal with this is one based on the radial scaling of the orbitals by suitable scale factors. In addition to very basic bonding arguments, it is soundly based on restoring the virial's theorem for finite bases, in the case of Coulombic potentials (all-electron calculations). The use of pseudopotentials limits its applicability, allowing only for extremely small deviations from unity ($\sim 1\%$) in the scale factors obtained variationally (with the exception of hydrogen that can contract up to 25%). This possiblity is available to the user.

Another way of dealing with the above problem and that of the kink at the same time is adding a soft confinement potential to the atomic Hamiltonian used to generate the basis orbitals: it smoothens the kink and contracts the orbital as suited. Two additional parameters are introduced for the purpose, which can be defined again variationally. The confining potential is flat (zero) in the core region, starts off at some internal radius r_i with all derivatives continuous and diverges at r_c ensuring the strict localization there. It is

$$V(r) = V_{\mathrm{o}} \frac{e^{-\frac{r_c - r_i}{r - r_i}}}{r_c - r} \tag{1}$$

and both r_i and V_{o} can be given to SIESTA together with r_c in the input (see **PAO.Basis** below). The kink is normally well smoothened with the default values for soft confinement by default (**PAO.SoftDefault** true), which are $r_i = 0.9\, r_c$ and $V_{\mathrm{o}} = 40$ Ry.

When explicitly introducing orbitals in the basis that would be empty in the atom (e.g. polarisation orbitals) these tend to be extremely extended if not completely unbound. The above procedure produces orbitals that bulge as far away from the nucleus as possible, to plunge abruptly at r_c. Soft confinement can be used to try to force a more reasonable shape, but it is not ideal (for orbitals peaking in the right region the tails tend to be far too short). *Charge confinement* produces very good shapes for empty orbitals. Essentially a Z/r potential is added to the soft confined potential above. For flexibility the charge confinement option in SIESTA is defined as

$$V_{\mathrm{Q}}(r) = \frac{Z e^{-\lambda r}}{\sqrt{r^2 + \delta^2}} \tag{2}$$

where δ is there to avoid the singularity (default $\delta = 0.01$ Bohr), and λ allows to screen the potential if longer tails are needed. The description on how to introduce this option can be found in the **PAO.Basis** entry below.

Finally, the shape of an orbital is also changed by the ionic character of the atom. Orbitals in cations tend to shrink, and they swell in anions. Introducing a δQ in the basis-generating free-atom calculations gives orbitals better adapted to ionic situations in the condensed systems.

More information about basis sets can be found in the proposed literature.

There are quite a number of options for the input of the basis-set and KB projector specification, and they are all optional! By default, SIESTA will use a DZP basis set with appropriate

choices for the determination of the range, etc. Of course, the more you experiment with the different options, the better your basis set can get. To aid in this process we offer an auxiliary program for optimization which can be used in particular to obtain variationally optimal basis sets (within a chosen basis size). See`Util/Optimizer` for general information, and `Util/Optimizer/Examples/Basis_Optim` for an example. The directory `Tutorials/Bases` in the main SIESTA DISTRIBUTION contains some tutorial material for the generation of basis sets and KB projectors.

Finally, some optimized basis sets for particular elements are available at the SIESTA web page. Again, it is the responsability of the users to test the transferability of the basis set to their problem under consideration.

6.3.2 Type of basis sets

PAO.BasisType (*string*):

The kind of basis to be generated is chosen. All are based on finite-range pseudo-atomic orbitals [PAO's of Sankey and Niklewsky, PRB 40, 3979 (1989)]. The original PAO's were described only for minimal bases. SIESTA generates extended bases (multiple-ζ, polarization, and diffuse orbitals) applying different schemes of choice:

- Generalization of the PAO's: uses the excited orbitals of the finite-range pseudo-atomic problem, both for multiple-ζ and for polarization [see Sánchez-Portal, Artacho, and Soler, JPCM **8**, 3859 (1996)]. Adequate for short-range orbitals.
- Multiple-ζ in the spirit of split valence, decomposing the original PAO in several pieces of different range, either defining more (and smaller) confining radii, or introducing Gaussians from known bases (Huzinaga's book).

All the remaining options give the same minimal basis. The different options and their FDF descriptors are the following:

- `split:` Split-valence scheme for multiple-zeta. The split is based on different radii.
- `splitgauss:` Same as `split` but using gaussian functions $e^{-(x/\alpha_i)^2}$. The gaussian widths α_i are read instead of the scale factors (see below). There is no cutting algorithm, so that a large enough r_c should be defined for the gaussian to have decayed sufficiently.
- `nodes:` Generalized PAO's.
- `nonodes:` The original PAO's are used, multiple-zeta is generated by changing the scale-factors, instead of using the excited orbitals.
- `filteret:` Use the filterets as a systematic basis set. The size of the basis set is controlled by the filter cut-off for the orbitals.

Note that, for the `split` and `nodes` cases the whole basis can be generated by SIESTA with no further information required. SIESTA will use default values as defined in the following (**PAO.BasisSize**, **PAO.EnergyShift**, and **PAO.SplitNorm**, see below).

Default value: `split`

6.3.3 Size of the basis set

PAO.BasisSize (*string*): It defines usual basis sizes. It has effect only if there is no block **PAO.Basis** present.

- `SZ` or `MINIMAL`: minimal or single-ζ basis.
- `DZ`: Double zeta basis, in the scheme defined by **PAO.BasisType**.
- `SZP`: Single-zeta basis plus polarization orbitals.
- `DZP` or `STANDARD`: Like `DZ` plus polarization orbitals. Polarization orbitals are constructed from perturbation theory, and they are defined so they have the minimum angular momentum l such that there are not occupied orbitals with the same l in the valence shell of the ground-state atomic configuration. They polarize the corresponding $l-1$ shell.

  Note: The ground-state atomic configuration used internally by SIESTA is defined in the source file `Src/periodic_table.f`. For some elements (e.g., Pd), the configuration might not be the standard one.

Default value: `DZP`

PAO.BasisSizes (*data block*): Block which allows to specify a different value of the variable PAO.BasisSize for each species. For example,

```
%block    PAO.BasisSizes
     Si       DZ
     H        DZP
     O        SZP
%endblock PAO.BasisSizes
```

6.3.4 Range of the orbitals

PAO.EnergyShift (*real energy*): A standard for orbital-confining cutoff radii. It is the excitation energy of the PAO's due to the confinement to a finite-range. It offers a general procedure for defining the confining radii of the original (first-zeta) PAO's for all the species guaranteeing the compensation of the basis. It only has an effect when the block **PAO.Basis** is not present or when the radii specified in that block are zero for the first zeta.

Use: It has to be positive.

Default value: `0.02 Ry`

6.3.5 Generation of multiple-zeta orbitals

PAO.SplitNorm (*real*): A standard to define sensible default radii for the split-valence type of basis. It gives the amount of norm that the second-ζ split-off piece has to carry. The split radius is defined accordingly. If multiple-ζ is used, the corresponding radii are obtained

by imposing smaller fractions of the SplitNorm (1/2, 1/4 ...) value as norm carried by the higher zetas. It only has an effect when the block **PAO.Basis** is not present or when the radii specified in that block are zero for zetas higher than one.

Default value: `0.15` (sensible values range between 0.05 and 0.5).

PAO.SplitNormH (*real*): This option is as per **PAO.SplitNorm** but allows a separate default to be specified for hydrogen which typically needs larger values than those for other elements.

PAO.NewSplitCode (*boolean*):

Enables a new, simpler way to match the multiple-zeta radii.

If an old-style (tail+parabola) calculation is being done, perform a scan of the tail+parabola norm in the whole range of the 1st-zeta orbital, and store that in a table. The construction of the 2nd-zeta orbital involves simply scanning the table to find the appropriate place. Due to the idiosyncracies of the old algorithm, the new one is not guaranteed to produce exactly the same results, as it might settle on a neighboring grid point for the matching.

Default value: `.false.`

PAO.FixSplitTable (*boolean*):

After the scan of the allowable split-norm values, apply a damping function to the tail to make sure that the table goes to zero at the radius of the first-zeta orbital.

Default value: `.false.`

PAO.SplitTailNorm (*boolean*):

Use the norm of the tail instead of the full tail+parabola norm. This is the behavior described in the JPC paper. (But note that, for numerical reasons, the square root of the tail norm is used in the algorithm.) This is the preferred mode of operation for automatic operation, as in non-supervised basis-optimization runs.

Default value: `.false.`

As a summary of the above options:

- For complete backwards compatibility, do nothing.
- To exercise the new code, set `PAO.NewSplitCode`.
- To maintain the old split-norm heuristic, but making sure that the program finds a solution (even if not optimal, in the sense of producing a second-ζ r_c very close to the first-ζ one), set `PAO.FixSplitTable` (this will automatically set `PAO.NewSplitCode`).
- If the old heuristic is of no interest (for example, if only a robust way of mapping split-norms to radii is needed), set `PAO.SplitTailNorm` (this will set `PAO.NewSplitCode` automatically).

PAO.EnergyCutoff (*real energy*): If the multiple zetas are generated using filterets then only the filterets with an energy lower than this cutoff are included. Increasing this value leads to a richer basis set (provided the cutoff is raised above the energy of any filteret that was

previously not included) but a more expensive calculation. It only has an effect when the option **PAO.BasisType** is set to filteret.

Use: It has to be positive.

Default value: `20.0 Ry`

PAO.EnergyPolCutoff (*real energy*): If the multiple zetas are generated using filterets then only the filterets with an energy lower than this cutoff are included for the polarisation functions. Increasing this value leads to a richer basis set (provided the cutoff is raised above the energy of any filteret that was previously not included) but a more expensive calculation. It only has an effect when the option **PAO.BasisType** is set to filteret.

Use: It has to be positive.

Default value: `20.0 Ry`

PAO.ContractionCutoff (*real*): If the multiple zetas are generated using filterets then any filterets that have a coefficient less than this threshold within the original PAO will be contracted together to form a single filteret. Increasing this value leads to a smaller basis set but allows the underlying basis to have a higher kinetic energy cut-off for filtering. It only has an effect when the option **PAO.BasisType** is set to filteret.

Use: It has to be in the range 0 to 1.

Default value: `0.0`

6.3.6 Soft-confinement options

PAO.SoftDefault (*boolean*): If set to true then this option causes soft confinement to be the default form of potential during orbital generation. The default potential and inner radius are set by the commands given below.

Default value: `.false.`

PAO.SoftInnerRadius (*real*): For default soft confinement, the inner radius is set at a fraction of the outer confinement radius determined by the energy shift. This option controls the fraction of the confinement radius to be used.

Default value: `0.9`

PAO.SoftPotential (*real*): For default soft confinement, this option controls the value of the potential used for all orbitals.

Default value: `40.0 Ry`

Note: Soft-confinement options (inner radius, prefactor) have been traditionally used to optimize the basis set, even though formally they are just a technical necessity to soften the decay of the orbitals at rc. To achieve this, it might be enough to use the above global options.

6.3.7 Kleinman-Bylander projectors

PS.lmax (*data block*): Block with the maximum angular momentum of the Kleinman-Bylander projectors, `lmxkb`. This information is optional. If the block is absent, or for a species which is not mentioned inside it, SIESTA will take `lmxkb(is) = lmxo(is) + 1`, where `lmxo(is)` is the maximum angular momentum of the basis orbitals of species `is`.

```
%block Ps.lmax
     Al_adatom   3
     H           1
     O           2
%endblock Ps.lmax
```

Default: (Block absent or empty). Maximum angular momentum of the basis orbitals plus one.

PS.KBprojectors (*data block*): This block provides information about the number of Kleinman-Bylander projectors per angular momentum, and for each species, that will used in the calculation. This block is optional. If the block is absent, or for species not mentioned in it, only one projector will be used for each angular momentum. The projectors will be constructed using the eigenfunctions of the respective pseudopotentials.

This block allows to specify the number of projector for each l, and also the reference energies of the wavefunctions used to build them. The specification of the reference energies is optional. If these energies are not given, the program will use the eigenfunctions with an increasing number of nodes (if there is not bound state with the corresponding number of nodes, the "eigenstates" are taken to be just functions which are made zero at very long distance of the nucleus). The units for the energy can be optionally specified, if not, the program will assumed that are given in Rydbergs. The data provided in this block must be consistent with those read from the block **PS.lmax**. For example,

```
%block PS.KBprojectors
    Si  3
     2   1
    -0.9     eV
     0   2
    -0.5  -1.0d4 Hartree
     1   2
    Ga  1
     1  3
    -1.0  1.0d5 -6.0
%endblock PS.KBprojectors
```

The reading is done this way (those variables in brackets are optional, therefore they are only read if present):

```
 From is = 1 to  nspecies
```

```
read: label(is), l_shells(is)
From lsh=1 to l_shells(is)
    read: l, nkbl(l,is)
    read: {erefKB(izeta,il,is)}, from ikb = 1 to nkbl(l,is), {units}
```

When a very high energy, higher that 1000 Ry, is specified, the default is taken instead. On the other hand, very low (negative) energies, lower than -1000 Ry, are used to indicate that the energy derivative of the last state must be used. For example, in the example given above, two projectors will be used for the *s* pseudopotential of Si. One generated using a reference energy of -0.5 Hartree, and the second one using the energy derivative of this state. For the *p* pseudopotential of Ga, three projectors will be used. The second one will be constructed from an automatically generated wavefunction with one node, and the other projectors from states at -1.0 and -6.0 Rydberg.

The analysis looking for possible *ghost* states is only performed when a single projector is used. Using several projectors some attention should be paid to the "KB cosine" (kbcos), given in the output of the program. The KB cosine gives the value of the overlap between the reference state and the projector generated from it. If these numbers are very small (< 0.01, for example) for **all** the projectors of some angular momentum, one can have problems related with the presence of ghost states.

Default: (Block absent or empty). Only one KB projector, constructed from the nodeless eigenfunction, used for each angular momentum.

KB.New.Reference.Orbitals (*boolean*):

If `.true.`, the routine to generate KB projectors will use slightly different parameters for the construction of the reference orbitals involved (Rmax=60 bohr both for integration and normalization).

Default value: `.false.`

6.3.8 The PAO.Basis block

PAO.Basis (*data block*): Block with data to define explicitly the basis to be used. It allows the definition by hand of all the parameters that are used to construct the atomic basis. There is no need to enter information for all the species present in the calculation. The basis for the species not mentioned in this block will be generated automatically using the parameters **PAO.BasisSize**, **PAO.BasisType**, **PAO.EnergyShift**, **PAO.SplitNorm** (or **PAO.SplitNormH**), and the soft-confinement defaults, if used (See **PAO.SoftDefault**).

Some parameters can be set to zero, or left out completely. In these cases the values will be generated from the magnitudes defined above, or from the appropriate default values. For example, the radii will be obtained from **PAO.EnergyShift** or from **PAO.SplitNorm** if they are zero; the scale factors will be put to 1 if they are zero or not given in the input. An example block for a two-species calculation (H and O) is the following (`opt` means optional):

```
%block PAO.Basis     # Define Basis set
```

```
O    2  nodes  1.0   # Label, l_shells, type (opt), ionic_charge (opt)
 n=2 0 2  E 50.0 2.5 # n (opt if not using semicore levels),l,Nzeta,Softconf(opt)
     3.50  3.50      #     rc(izeta=1,Nzeta)(Bohr)
     0.95  1.00      #     scaleFactor(izeta=1,Nzeta) (opt)
     1 1  P 2        # l, Nzeta, PolOrb (opt), NzetaPol (opt)
     3.50            #     rc(izeta=1,Nzeta)(Bohr)
H    2               # Label, l_shells, type (opt), ionic_charge (opt)
     0 2 S 0.2       # l, Nzeta, Per-shell split norm parameter
     5.00  0.00      #     rc(izeta=1,Nzeta)(Bohr)
     1 1 Q 3. 0.2    # l, Nzeta, Charge conf (opt): Z and screening
     5.00            #    rc(izeta=1,Nzeta)(Bohr)
%endblock PAO.Basis
```

The reading is done this way (those variables in brackets are optional, therefore they are only read if present) (See the routines in `Src/basis_specs.f` for detailed information):

```
From js = 1 to  nspecies
   read: label(is), l_shells(is), { type(is) }, { ionic_charge(is) }
   From lsh=1 to l_shells(is)
    read:
     { n }, l(lsh), nzls(lsh,is), { PolOrb(l+1) }, { NzetaPol(l+1) },
     {SplitNormfFlag(lsh,is)}, {SplitNormValue(lsh,is)}
     {SoftConfFlag(lsh,is)}, {PrefactorSoft(lsh,is)}, {InnerRadSoft(lsh,is)},
     {FilteretFlag(lsh,is)}, {FilteretCutoff(lsh,is)}
     {ChargeConfFlag(lsh,is)}, {Z(lsh,is)}, {Screen(lsh,is)}, {delta(lsh,is}
       read: rcls(izeta,lsh,is), from izeta = 1 to nzls(l,is)
       read: { contrf(izeta,il,is) }, from izeta = 1 to nzls(l,is)
```

And here is the variable description:

- `Label`: Species label, this label determines the species index `is` according to the block **ChemicalSpecieslabel**
- `l_shells(is)`: Number of shells of orbitals with different angular momentum for species `is`
- `type(is)`: *Optional input*. Kind of basis set generation procedure for species `is`. Same options as **PAO.BasisType**
- `ionic_charge(is)`: *Optional input*. Net charge of species `is`. This is only used for basis set generation purposes. *Default value*: `0.0` (neutral atom). Note that if the pseudopotential was generated in an ionic configuration, and no charge is specified in PAO.Basis, the ionic charge setting will be that of pseudopotential generation.
- `n`: Principal quantum number of the shell. This is an optional input for normal atoms, however it must be specified when there are *semicore* states (i.e. when states that usually are not considered to belong to the valence shell have been included in the calculation)
- `l`: Angular momentum of basis orbitals of this shell

- `nzls(lsh,is)`: Number of 'zetas' for this shell. For a filteret basis this number is ignored since the number is controlled by the cutoff.
- `PolOrb(l+1)`: *Optional input.* If set equal to `P`, a shell of polarization functions (with angular momentum $l + 1$) will be constructed from the first-zeta orbital of angular momentum l. *Default value*: ' ' (blank = No polarization orbitals).
- `NzetaPol(l+1)`: *Optional input.* Number of 'zetas' for the polarization shell (generated automatically in a split-valence fashion). For a filteret basis this number is ignored since the number is controlled by the cutoff. Only active if `PolOrb = P`. *Default value*: `1`
- `SplitNormFlag(lsh,is)`: *Optional input.* If set equal to `S`, the following number sets the split-norm parameter for that shell.
- `SoftConfFlag(l,is)`: *Optional input.* If set equal to `E`, the soft confinement potential proposed in equation (1) of the paper by J. Junquera *et al.*, Phys. Rev. B **64**, 235111 (2001), is used instead of the Sankey hard-well potential.
- `PrefactorSoft(l,is)`: *Optional input.* Prefactor of the soft confinement potential (V_0 in the formula). Units in Ry. *Default value*: 0 Ry.
- `InnerRadSoft(l,is)`: *Optional input.* Inner radius where the soft confinement potential starts off (r_i in the formula). If negative, the inner radius will be computed as the given fraction of the PAO cutoff radius. Units in bohrs. *Default value*: 0 bohrs.
- `FilteretFlag(l,is)`: *Optional input.* If set equal to `F`, then an individual filter cut-off can be specified for the shell.
- `FilteretCutoff(l,is)`: *Optional input.* Shell-specific value for the filteret basis cutoff. Units in Ry. *Default value*: The same as the value given by *FilterCutoff*.
- `ChargeConfFlag(lsh,is)`: *Optional input.* If set equal to `Q`, the charge confinement potential in equation (2) above is added to the confining potential. If present it requires at least one number after it (`Z`), but it can be followed by two or three numbers.
- `Z(lhs,is)`: *Optional input, needed if* `Q` *is set.* Z charge in equation (2) above for charge confinement (units of e).
- `Screen(lhs,is)`: *Optional input.* Yukawa screening parameter λ in equation (2) above for charge confinement (in Bohr^{-1}).
- `delta(lhs,is)`: *Optional input.* Singularity regularisation parameter δ in equation (2) above for charge confinement (in Bohr).
- `rcls(izeta,l,is)`: Cutoff radius (Bohr) of each 'zeta' for this shell. For the second zeta onwards, if this value is negative, the actual rc used will be the given fraction of the first zeta's rc.
- `contrf(izeta,l,is)`: *Optional input.* Contraction factor of each 'zeta' for this shell. *Default value*: `1.0`

Polarization orbitals are generated by solving the atomic problem in the presence of a polarizing electric field. The orbitals are generated applying perturbation theory to the first-zeta orbital of lower angular momentum. They have the same cutoff radius as the orbitals from which they are constructed.

Note: The perturbative method has traditionally used the 'l' component of the pseudopotential. It can be argued that it should use the 'l+1' component. By default, for backwards compatibility, the traditional method is used, but the alternative one can be activated by setting the logical **PAO.OldStylePolOrbs** variable to `.false.`

There is a different possibility for generating polarization orbitals: by introducing them explicitly in the **PAO.Basis** block. It has to be remembered, however, that they sometimes correspond to unbound states of the atom, their shape depending very much on the cutoff radius, not converging by increasing it, similarly to the multiple-zeta orbitals generated with the `nodes` option. Using **PAO.EnergyShift** makes no sense, and a cut off radius different from zero must be explicitly given (the same cutoff radius as the orbitals they polarize is usually a sensible choice).

A species with atomic number = -100 will be considered by SIESTA as a constant-pseudopotential atom, *i.e.*, the basis functions generated will be spherical Bessel functions with the specified r_c. In this case, r_c has to be given, as **EnergyShift** will not calculate it.

Other negative atomic numbers will be interpreted by SIESTA as *ghosts* of the corresponding positive value: the orbitals are generated and put in position as determined by the coordinates, but neither pseudopotential nor electrons are considered for that ghost atom. Useful for BSSE correction.

Use: This block is optional, except when Bessel functions or semicore states are present.

Default: Basis characteristics defined by global definitions given above.

6.3.9 Filtering

FilterCutoff (*physical energy*): Kinetic energy cutoff of plane waves used to filter all the atomic basis functions, the pseudo-core densities for partial core corrections, and the neutral-atom potentials. The basis functions (which must be squared to obtain the valence density) are really filtered with a cutoff reduced by an empirical factor $0.7^2 \simeq 0.5$. The **FilterCutoff** should be similar or lower than the **MeshCutoff** to avoid the *eggbox effect* on the atomic forces. However, one should not try to converge **MeshCutoff** while simultaneously changing **FilterCutoff**, since the latter in fact changes the used basis functions. Rather, fix a sufficiently large **FilterCutoff** and converge only **MeshCutoff**. If **FilterCutoff** is not explicitly set, its value is calculated from **FilterTol**.

FilterTol (*physical energy*): Residual kinetic-energy leaked by filtering each basis function. While **FilterCutoff** sets a common reciprocal-space cutoff for all the basis functions, **FilterTol** sets a specific cutoff for each basis function, much as the **EnergyShift** sets their real-space cutoff. Therefore, it is reasonable to use similar values for both parameters. The maximum cutoff required to meet the **FilterTol**, among all the basis functions, is used (multiplied by the empirical factor $1/0.7^2 \simeq 2$) to filter the pseudo-core densities and the neutral-atom potentials. **FilterTol** is ignored if **FilterCutoff** is present in the input file. If neither **FilterCutoff** nor **FilterTol** are present, no filtering is performed. See Soler and Anglada, arXiv:0807.5030, for details of the filtering procedure.

Warning: If the value of **FilterCutoff** is made too small (or **FilterTol** too large) some of the filtered basis orbitals may be meaningless, leading to incorrect results or even a program crash.

To be implemented: If **MeshCutoff** is not present in the input file, it can be set using the maximum filtering cutoff used for the given **FilterTol** (for the time being, you can use `AtomSetupOnly T` to stop the program after basis generation, look at the maximum filtering cutoff used, and set the mesh-cutoff manually in a later run.)

6.3.10 Saving and reading basis-set information

SIESTA (and the standalone program GEN-BASIS) always generate the files *Atomlabel*`.ion`, where *Atomlabel* is the atomic label specified in block *ChemicalSpeciesLabel*. Optionally, if NetCDF support is compiled in, the programs generate NetCDF files *Atomlabel*`.ion.nc` (except for ghost atoms). See an Appendix for information on the optional NetCDF package.

These files can be used to read back information into SIESTA.

User.Basis (*logical*):

If true, the basis, KB projector, and other information is read from files *Atomlabel*`.ion`, where *Atomlabel* is the atomic species label specified in block *ChemicalSpeciesLabel*. These files can be generated by a previous SIESTA run or (one by one) by the standalone program GEN-BASIS. No pseudopotential files are necessary.

User.Basis.NetCDF (*logical*):

If true, the basis, KB projector, and other information is read from NetCDF files *Atomlabel*`.ion.nc`, where *Atomlabel* is the atomic label specified in block *ChemicalSpeciesLabel*. These files can be generated by a previous SIESTA run or by the standalone program GEN-BASIS. No pseudopotential files are necessary. NetCDF support is needed. Note that ghost atoms cannot yet be adequately treated with this option.

6.3.11 Tools to inspect the orbitals and KB projectors

The program `ioncat` in `Util/Gen-basis` can be used to extract orbital, KB projector, and other information contained in the `.ion` files. The output can be easily plotted with a graphics program. If the option **WriteIonPlotFiles** is enabled, SIESTA will generate and extra set of files that can be plotted with the `gnuplot` scripts in `Tutorials/Bases`. The stand-alone program `gen-basis` sets that option by default, and the script `Tutorials/Bases/gen-basis.sh` can be used to automate the process. See also the NetCDF-based utilities in `Util/PyAtom`.

6.3.12 Basis optimization

There are quite a number of options for the input of the basis-set and KB projector specification, and they are all optional! By default, SIESTA will use a DZP basis set with appropriate choices for the determination of the range, etc. Of course, the more you experiment with

the different options, the better your basis set can get. To aid in this process we offer an auxiliary program for optimization which can be used in particular to obtain variationally optimal basis sets (within a chosen basis size). See`Util/Optimizer` for general information, and `Util/Optimizer/Examples/Basis_Optim` for an example.

BasisPressure (*real pressure*):

SIESTA will compute and print the value of the "effective basis enthalpy" constructed by adding a term of the form $p_{basis}V_{orbs}$ to the total energy. Here p_{basis} is a fictitious basis pressure and V_{orbs} is the volume of the system's orbitals. This is a useful quantity for basis optimization (See Anglada *et al.*). The total basis enthalpy is also written to the ASCII file `BASIS_ENTHALPY`.

Default value: 0.2 GPa

6.3.13 Low-level options regarding the radial grid

For historical reasons, the basis-set and KB projector code in SIESTA uses a logarithmic radial grid, which is taken from the pseudopotential file. Any "interesting" radii have to fall on a grid point, which introduces a certain degree of coarseness that can limit the accuracy of the results and the faithfulness of the mapping of input parameters to actual operating parameters. For example, the same orbital will be produced by a finite range of **PAO.EnergyShift** values, and any user-defined cutoffs will not be exactly reflected in the actual cutoffs. This is particularly troublesome for automatic optimization procedures (such as those implemented in `Util/Optimizer`), as the engine might be confused by the extra level of indirection. The following options can be used to fine-tune the mapping. They are not enabled by default, as they change the numerical results apreciably (in effect, they lead to different basis orbitals and projectors).

Reparametrize.Pseudos (*logical*):

By changing the a and b parameters of the logarithmic grid, a new one with a more adequate grid-point separation can be used for the generation of basis sets and projectors. For example, by using $a = 0.001$ and $b = 0.01$, the grid point separations at $r = 0$ and 10 bohrs are 0.00001 and 0.01 bohrs, respectively. More points are needed to reach r's of the order of a hundred bohrs, but the extra computational effort is negligible. The net effect of this option (notably when coupled to **Restricted.Radial.Grid** `.false.`) is a closer mapping of any user-specified cutoff radii and of the radii implicitly resulting from other input parameters to the actual values used by the program. (The small grid-point separation near r=0 is still needed to avoid instabilities for s channels that occurred with the previous (reparametrized) default spacing of 0.005 bohr. This effect is not yet completely understood.)

Default value: `.false.`

New.A.Parameter (*real*):

New setting for the pseudopotential grid's a parameter

Default value: 0.001

New.B.Parameter (*real*):

New setting for the pseudopotential grid's b parameter

Default value: 0.01

Rmax.Radial.Grid (*real*):

New setting for the maximum value of the radial coordinate for integration of the atomic Schrodinger equation.

Default value: 50.0 if `Reparametrize.Pseudos` is set, zero otherwise (which means that the maximum radius from the pseudopotential file is used).

Restricted.Radial.Grid (*logical*):

In normal operation of the basis-set and projector generation code the radial grid is restricted to having an odd number of points, and the cutoffs are shifted accordingly. This restriction can be lifted by setting this parameter to `.false.`

Default value: `.true.`

6.4 Structural information

There are many ways to give SIESTA structural information.

- Directly from the fdf file in traditional format.
- Directly from the fdf file in the newer Z-Matrix format, using a **Zmatrix** block.
- From an external data file
- From a FIFO file, when working in "server" mode.

Note that, regardless of the way in which the structure is described, the tags **NumberOfSpecies**, **NumberOfAtoms**, and **ChemicalSpeciesLabel** are mandatory.

In the following sections we document the different structure input methods, and provide a guide to their precedence.

6.4.1 Traditional structure input in the fdf file

Firstly, the size of the cell itself should be specified, using some combination of the options **LatticeConstant**, **LatticeParameters**, and **LatticeVectors**, and **SuperCell**. If nothing is specified, SIESTA will construct a cubic cell in which the atoms will reside as a cluster.

Secondly, the positions of the atoms within the cells must be specified, using either the traditional SIESTA input format (a modified xyz format) which must be described within a **AtomicCoordinatesAndAtomicSpecies** block.

LatticeConstant (*real length*): Lattice constant. This is just to define the scale of the lattice vectors.

Default value: Minimum size to include the system (assumed to be a molecule) without intercell interactions, plus 10%.

NOTE: A LatticeConstant value, even if redundant, might be needed for other options, such as the units of the k-points used for band-structure calculations. This mis-feature will be corrected in future versions.

LatticeParameters (*data block*): Crystallographic way of specifying the lattice vectors, by giving six real numbers: the three vector modules, a, b, and c, and the three angles α (angle between $\vec{b}$ and $\vec{c}$), β, and γ. The three modules are in units of **LatticeConstant**, the three angles are in degrees.

Default value:

```
1.0   1.0   1.0    90.   90.  90.
```

(see the following)

LatticeVectors (*data block*): The cell vectors are read in units of the lattice constant defined above. They are read as a matrix `CELL(ixyz,ivector)`, each vector being one line.

Default value:

```
1.0    0.0    0.0
0.0    1.0    0.0
0.0    0.0    1.0
```

If the **LatticeConstant** default is used, the default of **LatticeVectors** is still diagonal but not necessarily cubic.

SuperCell (*data block*): Integer 3x3 matrix defining a supercell in terms of the unit cell:

```
%block SuperCell
   M(1,1)  M(2,1)  M(3,1)
   M(1,2)  M(2,2)  M(3,2)
   M(1,3)  M(2,3)  M(3,3)
%endblock SuperCell
```

and the supercell is defined as $SuperCell(ix, i) = \sum_j CELL(ix, j) * M(j, i)$. Notice that the matrix indexes are inverted: each input line specifies one supercell vector.

Warning: **SuperCell** is disregarded if the geometry is read from the XV file, which can happen unadvertedly.

Use: The atomic positions must be given only for the unit cell, and they are 'cloned' automatically in the rest of the supercell. The **NumberOfAtoms** given must also be that in a single unit cell. However, all values in the output are given for the entire

supercell. In fact, CELL is inmediately redefined as the whole supercell and the program no longer knows the existence of an underlying unit cell. All other input (apart from NumberOfAtoms and atomic positions), including **kgridMonkhorstPack** must refer to the supercell (this is a change over previous versions). Therefore, to avoid confusions, we recommend to use **SuperCell** only to generate atomic positions, and then to copy them from the output to a new input file with all the atoms specified explicitly and with the supercell given as a normal unit cell.

Default value: No supercell (supercell equal to unit cell).

AtomicCoordinatesFormat (*string*): Character string to specify the format of the atomic positions in input. These can be expressed in four forms:

- `Bohr` or `NotScaledCartesianBohr` (atomic positions are given directly in Bohr, in Cartesian coordinates)
- `Ang` or `NotScaledCartesianAng` (atomic positions are given directly in Ångström, in Cartesian coordinates)
- `ScaledCartesian` (atomic positions are given in Cartesian coordinates, in units of the lattice constant)
- `Fractional` or `ScaledByLatticeVectors` (atomic positions are given referred to the lattice vectors)

Default value: `NotScaledCartesianBohr`

AtomCoorFormatOut (*string*): Character string to specify the format of the atomic positions in output. Same possibilities as for input (**AtomicCoordinatesFormat**).

Default value: value of **AtomicCoordinatesFormat**

AtomicCoordinatesOrigin (*data block*): Vector specifying a rigid shift to apply to the atomic coordinates, given in the same format and units as these. Notice that the atomic positions (shifted or not) need not be within the cell formed by **LatticeVectors**, since periodic boundary conditions are always assumed.

Default value:

```
0.000   0.000   0.000
```

AtomicCoordinatesAndAtomicSpecies (*data block*): Block specifying the position and species of each atom. One line per atom, the reading is done this way:

```
From ia = 1 to natoms
     read: xa(ix,ia), isa(ia)
```

where `xa(ix,ia)` is the `ix` coordinate of atom `iai` in the format (units) specified by **AtomCoordinatesFormat**, and `isa(ia)` is the species index of atom `ia`.

Default: There is no default. The positions must be introduced either using this block or the Z matrix (see **Zmatrix**).

6.4.2 Z-matrix format and constraints

The advantage of the traditional format is that it is much easier to set up a system. However, when working on systems with constraints, there are only a limited number of (very simple) constraints that may be expressed within this format, and recompilation is needed for each new constraint.

For any more involved set of constraints, a full **Zmatrix** formulation should be used - this offers much more control, and may be specified fully at run time (thus not requiring recompilation) - but it is more work to generate the input files for this form.

Zmatrix (*data block*): This block provides a means for inputting the system geometry using a Z-matrix format, as well as controlling the optimization variables. This is particularly useful when working with molecular systems or restricted optimizations (such as locating transition states or rigid unit movements). The format also allows for hybrid use of Z-matrices and Cartesian or fractional blocks, as is convenient for the study of a molecule on a surface. As is always the case for a Z-matrix, the responsibility falls to the user to chose a sensible relationship between the variables to avoid triads of atoms that become linear.

Below is an example of a Z-matrix input for a water molecule:

```
%block Zmatrix

molecule fractional
  1 0 0 0   0.0 0.0 0.0 0 0 0
  2 1 0 0   HO1 90.0 37.743919 1 0 0
  2 1 2 0   HO2 HOH 90.0 1 1 0
variables
    HO1 0.956997
    HO2 0.956997
    HOH 104.4
%endblock Zmatrix
```

The sections that can be used within the Zmatrix block are as follows:

Firstly, all atomic positions must be specified within either a "`molecule`" block or a "`cartesian`" block. Any atoms subject to constraints more complicated than "do not change this coordinate of this atom" must be specified within a "`molecule`" block.

`molecule`:

There must be one of these blocks for each independent set of constrained atoms within the simulation.

This specifies the atoms that make up each molecule and their geometry. In addition, an option of "`fractional`" or "`scaled`" may be passed, which indicates that distances are specified in scaled or fractional units. In the absence of such an option, the distance units are taken to be the value of "`ZM.UnitsLength`".

A line is needed for each atom in the molecule; the format of each line should be:

```
Nspecies i j k r a t ifr ifa ift
```

Here the values `Nspecies`, `i`, `j`, `k`, `ifr`, `ifa`, and `ift` are integers and `r`, `a`, and `t` are double precision reals.

For most atoms, `Nspecies` is the species number of the atom, `r` is distance to atom number `i`, `a` is the angle made by the present atom with atoms `j` and `i`, while `t` is the torsional angle made by the present atom with atoms `k`, `j`, and `i`. The values `ifr`, `ifa` and `ift` are integer flags that indicate whether `r`, `a`, and `t`, respectively, should be varied; 0 for fixed, 1 for varying.

The first three atoms in a molecule are a special case. Because there are insufficient atoms defined to specify a distance/angle/torsion, the values are set differently. For atom 1, `r`, `a`, and `t`, are the Cartesian coordinates of the atom. For the second atom, `r`, `a`, and `t` are the coordinates in spherical form of the second atom relative to the first: first the radius, then the polar angle (angle between the z-axis and the displacement vector) and then the azimuthal angle (angle between the x-axis and the projection of the displacement vector on the x-y plane). Finally, for the third atom, the numbers take their normal form, but the torsional angle is defined relative to a notional atom 1 unit in the z-direction above the atom `j`.

Secondly. blocks of atoms all of which are subject to the simplest of constraints may be specified in one of the following three ways, according to the units used to specify their coordinates:

`cartesian`: This section specifies a block of atoms whose coordinates are to be specified in Cartesian coordinates. Again, an option of "`fractional`" or "`scaled`" may be added, to specify the units used; and again, in their absence, the value of "`ZM.UnitsLength`" is taken.

The format of each atom in the block will look like:

```
Nspecies x y z ix iy iz
```

Here `Nspecies`, `ix`, `iy`, and `iz` are integers and `x`, `y`, `z` are reals. `Nspecies` is the species number of the atom being specified, while `x`, `y`, and `z` are the Cartesian coordinates of the atom in whichever units are being used. The values `ix`, `iy` and `iz` are integer flags that indicate whether the `x`, `y`, and `z` coordinates, respectively, should be varied or not. A value of 0 implies that the coordinate is fixed, while 1 implies that it should be varied. **NOTE**: When performing "variable cell" optimization while using a Zmatrix format for input, the algorithm will not work if some of the coordinates of an atom in a `cartesian` block are variables and others are not (i.e., `ix iy iz` above must all be 0 or 1). This will be fixed in future versions of the program.

A Zmatrix block may also contain the following, additional, sections, which are designed to make it easier to read.

`constants`: Instead of specifying a numerical value, it is possible to specify a symbol within the above geometry definitions. This section allows the user to define the value of the symbol as a constant. The format is just a symbol followed by the value:

```
HOH 104.4
```

`variables`: Instead of specifying a numerical value, it is possible to specify a symbol within the above geometry definitions. This section allows the user to define the value of the symbol as a variable. The format is just a symbol followed by the value:

```
HO1 0.956997
```

Finally, constraints must be specified in a `constraints` block.

`constraint` This sub-section allows the user to create constraints between symbols used in a Z-matrix:

```
constraint
  var1 var2 A B
```

Here var1 and var2 are text symbols for two quantities in the Z-matrix definition, and A and B are real numbers. The variables are related by $var1 = A * var2 + B$.

An example of a Z-matrix input for a benzene molecule over a metal surface is:

```
%block Zmatrix
molecule
  2 0 0 0 xm1 ym1 zm1 0 0 0
  2 1 0 0 CC 90.0 60.0 0 0 0
  2 2 1 0 CC CCC 90.0 0 0 0
  2 3 2 1 CC CCC 0.0 0 0 0
  2 4 3 2 CC CCC 0.0 0 0 0
  2 5 4 3 CC CCC 0.0 0 0 0
  1 1 2 3 CH CCH 180.0 0 0 0
  1 2 1 7 CH CCH 0.0 0 0 0
  1 3 2 8 CH CCH 0.0 0 0 0
  1 4 3 9 CH CCH 0.0 0 0 0
  1 5 4 10 CH CCH 0.0 0 0 0
  1 6 5 11 CH CCH 0.0 0 0 0
fractional
  3 0.000000 0.000000 0.000000 0 0 0
  3 0.333333 0.000000 0.000000 0 0 0
  3 0.666666 0.000000 0.000000 0 0 0
  3 0.000000 0.500000 0.000000 0 0 0
  3 0.333333 0.500000 0.000000 0 0 0
  3 0.666666 0.500000 0.000000 0 0 0
  3 0.166667 0.250000 0.050000 0 0 0
  3 0.500000 0.250000 0.050000 0 0 0
  3 0.833333 0.250000 0.050000 0 0 0
  3 0.166667 0.750000 0.050000 0 0 0
  3 0.500000 0.750000 0.050000 0 0 0
  3 0.833333 0.750000 0.050000 0 0 0
```

```
  3 0.000000 0.000000 0.100000 0 0 0
  3 0.333333 0.000000 0.100000 0 0 0
  3 0.666666 0.000000 0.100000 0 0 0
  3 0.000000 0.500000 0.100000 0 0 0
  3 0.333333 0.500000 0.100000 0 0 0
  3 0.666666 0.500000 0.100000 0 0 0
  3 0.166667 0.250000 0.150000 0 0 0
  3 0.500000 0.250000 0.150000 0 0 0
  3 0.833333 0.250000 0.150000 0 0 0
  3 0.166667 0.750000 0.150000 0 0 0
  3 0.500000 0.750000 0.150000 0 0 0
  3 0.833333 0.750000 0.150000 0 0 0
constants
    ym1 3.68
variables
    zm1 6.9032294
    CC 1.417
    CH 1.112
    CCH 120.0
    CCC 120.0
constraints
    xm1 CC -1.0 3.903229
%endblock Zmatrix
```

Here the species 1, 2 and 3 represent H, C, and the metal of the surface, respectively.

(Note: the above example shows the usefulness of symbolic names for the relevant coordinates, in particular for those which are allowed to vary. The current output options for Zmatrix information work best when this approach is taken. By using a "fixed" symbolic Zmatrix block and specifying the actual coordinates in a "variables" section, one can monitor the progress of the optimization and easily reconstruct the coordinates of intermediate steps in the original format.)

Use: Specifies the geometry of the system according to a Z-matrix *Default value:* Geometry is not specified using a Z-matrix

ZM.UnitsLength (*length*): Parameter that specifies the units of length used during Z-matrix input.

Use: This option allows the user to chose between inputing distances in Bohr or Angstroms within the Z-matrix data block.

Default value: `Bohr`

ZM.UnitsAngle (*angle*): Parameter that specifies the units of angles used during Z-matrix input.

Use: This option allows the user to chose between inputing angles in radians or degrees within the Z-matrix data block.

Default value: `rad`

6.4.3 Output of structural information

SIESTA is able to generate several kinds of files containing structural information (maybe too many).

- **STRUCT_OUT file:** Siesta always produces a `.STRUCT_OUT` file with cell vectors in Å and atomic positions in fractional coordinates. This file, renamed to *SystemLabel*.STRUCT_IN can be used for crystal-structure input. Note that the geometry reported is the last one for which forces and stresses were computed. See **MD.UseStructFile**.

- **STRUCT_NEXT_ITER file:** This file is always written, in the same format as `.STRUCT_OUT` file. The only difference is that it contains the structural information *after* it has been updated by the relaxation or the molecular-dynamics algorithms, and thus it could be used as input (renamed as *SystemLabel*.STRUCT_IN) for a continuation run, in the same way as the `XV` file.

  See **MD.UseStructFile**.

- **XV file:** The coordinates are always written in the *Systemlabel*.XV file, and overriden at every step.

- **OUT.UCELL.ZMATRIX file:** @*OUT.UCELL.ZMATRIX*

  This file is produced if the Zmatrix format is being used for input. (Please note that *SystemLabel* is not used as a prefix.) It contains the structural information in fdf form, with blocks for unit-cell vectors and for Zmatrix coordinates. The Zmatrix block is in a "canonical" form with the following characteristics:

  ```
  1. No symbolic variables or constants are used.
  2. The position coordinates of the first atom in each molecule
     are absolute Cartesian coordinates.
  3. Any coordinates in ''cartesian'' blocks are also absolute Cartesians.
  4. There is no provision for output of constraints.
  5. The units used are those initially specified by the user, and are
     noted also in fdf form.
  ```

  Note that the geometry reported is the last one for which forces and stresses were computed.

- **NEXT_ITER.UCELL.ZMATRIX file:** @*NEXT_ITER.UCELL.ZMATRIX*

  A file with the same format as `OUT.UCELL.ZMATRIX` but with a possibly updated geometry.

- The coordinates can be also accumulated in the *Systemlabel*.MD or *Systemlabel*.MDX files depending on **WriteMDhistory**.

- Additionally, several optional formats are supported:

WriteCoorXmol (*logical*): If `.true.` it originates the writing of an extra file named *SystemLabel*`.xyz` containing the final atomic coordinates in a format directly readable by XMol.[1] Coordinates come out in Ångström independently of what specified in **AtomicCoordinatesFormat** and in **AtomCoorFormatOut**. There is a present Java implementation of XMol called JMol.

Default value: `.false.`

WriteCoorCerius (*logical*): If `.true.` it originates the writing of an extra file named *SystemLabel*`.xtl` containing the final atomic coordinates in a format directly readable by Cerius.[2] Coordinates come out in `Fractional` format (the same as `ScaledByLatticeVectors`) independently of what specified in **AtomicCoordinatesFormat** and in **AtomCoorFormatOut**. If negative coordinates are to be avoided, it has to be done from the start by shifting all the coordinates rigidly to have them positive, by using **AtomicCoordinatesOrigin**. See the Sies2arc utility in the Util/ directory for generating .arc files for CERIUS animation.

Default value: `.false.`

WriteMDXmol (*logical*): If `.true.` it causes the writing of an extra file named *SystemLabel*`.ANI` containing all the atomic coordinates of the simulation in a format directly readable by XMol for animation. Coordinates come out in Ångström independently of what is specified in **AtomicCoordinatesFormat** and in **AtomCoorFormatOut**. This file is accumulative even for different runs.

There is an alternative for animation by generating a .arc file for CERIUS. It is through the Sies2arc postprocessing utility in the Util/ directory, and it requires the coordinates to be accumulated in the output file, i.e., WriteCoorStep = `.true.`

Default value: `.false.`

Note change with respect to previous versions. This option is no longer coupled to **WriteCoorStep**.

6.4.4 Input of structural information from external files

The structural information can be also read from external files. Note that the NumberOfAtoms, NumberOfSpecies, and ChemicalSpeciesLabel options are still mandatory in the fdf file.

- The XV file. The logical variable **MD.UseSaveXV** instructs Siesta to read the atomic positions and velocities stored in file `SystemLabel.XV` by a previous run.

  If the required file does not exist, a warning is printed but the program does not stop. Overrides **UseSaveData**, but can be implicitly set by it.

  Default value: `.false.`

- A .STRUCT_IN file. The logical fdf variable **UseStructFile** (for historical reasons, **MD.UseStructFile** is also accepted, but deprecated) controls whether the structural

[1] XMol is under © copyright of Research Equipment Inc., dba Minnesota Supercomputer Center Inc.
[2] Cerius is under © copyright of Molecular Simulations Inc.

information is read from an external file of name *SystemLabel*.STRUCT_IN. If `.true.`, all other structural information in the fdf file will be ignored.

The format of the file is implied by the following code:

```
read(*,*) ((cell(ixyz,ivec),ixyz=1,3),ivec=1,3)  ! Cell vectors, in Angstroms
read(*,*) na
do ia = 1,na
   read(iu,*) isa(ia), dummy, xfrac(1:3,ia)  ! Species number
                                             ! Dummy numerical column
                                             ! Fractional coordinates
enddo
```

Warning: Note that the resulting geometry could be clobbered if an XV file is read after this file. It is up to the user to remove any XV files..

Default value: `.false.`

- A Zmatrix input file **MD.UseSaveZM** (*logical*): instructs the program to read the Zmatrix information stored in file `SystemLabel`.ZM by a previous run.

  Use: If the required file does not exist, a warning is printed but the program does not stop. Overrides **UseSaveData**, but can be implicitly set by it.

  Warning: Note that the resulting geometry could be clobbered if an XV file is read after this file. It is up to the user to remove any XV files..

  Default value: `.false.`

6.4.5 Input from a FIFO file

See the "Forces" option in **MD.TypeOfRun**. Note that the NumberOfAtoms, NumberOfSpecies, and ChemicalSpeciesLabel options are still mandatory in the fdf file.

6.4.6 Precedence issues in structural input

- If the "server" option is active, it takes precedence over everything (it will overwrite all other input with the information it gets from the FIFO file).
- If MD.UseSaveXV is active, it takes precedence over the options below.
- If UseStructFile (or MD.UseStructFile) is active, it takes precedence over the options below.
- For atomic coordinates, the traditional and Zmatrix formats in the fdf file are mutually exclusive. If **MD.UseSaveZM** is active, the contents of the ZM file, if found, take precedence over the Zmatrix information in the fdf file.

6.4.7 Interatomic distances

WarningMinimumAtomicDistance (*physical*): Fixes a threshold interatomic distance below which a warning message is printed.

Default value: `1.0 Bohr`

MaxBondDistance (*physical*): SIESTA prints the interatomic distances, up to a range of `MaxBondDistance`, to file `SystemLabel.BONDS` upon first reading the structural information, and to file `SystemLabel.BONDS_FINAL` after the last geometry iteration. The reference atoms are all the atoms in the unit cell. The routine now prints the real location of the neighbor atoms in space, and not, as in earlier versions, the location of the equivalent representative in the unit cell.

Default value: `6.0 Bohr`

6.5 *k*-point sampling

These are options for the k-point grid used in the SCF cycle. For other specialized grids, see the Macroscopic Polarization and Density of States sections.

kgrid_cutoff (*real length*): Parameter which determines the fineness of the k-grid used for Brillouin zone sampling. It is half the length of the smallest lattice vector of the supercell required to obtain the same sampling precision with a single k point. Ref: Moreno and Soler, PRB 45, 13891 (1992).

Use: If it is zero, only the gamma point is used. The resulting k-grid is chosen in an optimal way, according to the method of Moreno and Soler (using an effective supercell which is as spherical as possible, thus minimizing the number of k-points for a given precision). The grid is displaced for even numbers of effective mesh divisions. This parameter is not used if **kgrid_Monkhorst_Pack** is specified. If the unit cell changes during the calculation (for example, in a cell-optimization run, the k-point grid will change accordingly (see **ChangeKgridInMD** for the case of variable-cell molecular-dynamics runs, such as Parrinello-Rahman). This is analogous to the changes in the real-space grid, whose fineness is specified by an energy cutoff. If sudden changes in the number of k-points are not desired, then the Monkhorst-Pack data block should be used instead. In this case there will be an implicit change in the quality of the sampling as the cell changes. Both methods should be equivalent for a well-converged sampling.

Default value: `0.0 Bohr`

kgrid_Monkhorst_Pack (*data block*): Real-space supercell, whose reciprocal unit cell is that of the k-sampling grid, and grid displacement for each grid coordinate. Specified as an integer matrix and a real vector:

```
%block kgrid_Monkhorst_Pack
   Mk(1,1)  Mk(2,1)  Mk(3,1)   dk(1)
   Mk(1,2)  Mk(2,2)  Mk(3,2)   dk(2)
```

```
   Mk(1,3)  Mk(2,3)  Mk(3,3)   dk(3)
%endblock kgrid_Monkhorst_Pack
```

where `Mk(j,i)` are integers and `dk(i)` are usually either 0.0 or 0.5 (the program will warn the user if the displacements chosen are not optimal). The k-grid supercell is defined from `Mk` as in block **SuperCell** above, i.e.: $KgridSuperCell(ix,i) = \sum_j CELL(ix,j) * Mk(j,i)$. Note again that the matrix indexes are inverted: each input line gives the decomposition of a supercell vector in terms of the unit cell vectors.

Use: Used only if **SolutionMethod** = `diagon`. The k-grid supercell is compatible and unrelated (except for the default value, see below) with the **SuperCell** specifier. Both supercells are given in terms of the CELL specified by the **LatticeVectors** block. If `Mk` is the identity matrix and `dk` is zero, only the Γ point of the **unit** cell is used. Overrides **kgrid_cutoff**

Default value: Γ point of the (super)cell. (Default used only when **kgrid_cutoff** is not defined).

ChangeKgridInMD (*boolean*):

If `.true.`, the k-point grid is recomputed at every iteration during MD runs that potentially change the unit cell: Parrinello-Rahman, Nose-Parrinello-Rahman, and Anneal. Regardless of the setting of this flag, the k-point grid is always updated at every iteration of a variable-cell optimization and after each step in a "siesta-as-server" run.

Default value: `.false.` for historical reasons. The rationale was to avoid sudden jumps in some properties when the sampling changes, but if the calculation is well-converged there should be no problems if the update is enabled.

TimeReversalSymmetryForKpoints (*boolean*):

If `.true.`, the k-points in the BZ generated by the methods above are paired as (k,$-k$) and only one member of the pair is retained. This symmetry is valid in the absence of external magnetic fields or non-collinear/spin-orbit interaction.

This flag should be used with care, as the code will produce wrong results if there is no support for the appropriate symmetrization.

Default value: `.true.` unless: a) the option **SpinSpiral** is used. In this case time-reversal-symmetry is broken explicitly. b) non-collinear spin calculations. This case is less clear cut, but the time-reversal symmetry is not used to avoid possible breakings due to subtle implementation details, and to make the set of wavefunctions compatible with spin-orbit-capable versions (¿4.1) in analysis tools.

6.5.1 Output of k-point information

The coordinates of the $\vec{k}$ points used in the sampling are always stored in the file SystemLabel.KP.

WriteKpoints (*logical*): If `.true.` it writes the coordinates of the $\vec{k}$ vectors used in the grid for k-sampling, into the main output file.

Default value: `.false.` (see **LongOutput**)

6.6 Exchange-correlation functionals

XC.functional (*string*): Exchange-correlation functional type. May be `LDA` (local density approximation, equivalent to `LSD`), `GGA` (Generalized Gradient Approximation), or `VDW` (van der Waals).

Use: Spin polarization is defined by SpinPolarized label for both `LDA` and `GGA`. There is no difference between `LDA` and `LSD`.

Default value: `LDA`

XC.authors (*string*): Particular parametrization of the exchange-correlation functional. Options are:

- `CA` (equivalent to `PZ`): (Spin) local density approximation (LDA/LSD). Quantum Monte Carlo calculation of the homogeneous electron gas by D. M. Ceperley and B. J. Alder, Phys. Rev. Lett. **45**,566 (1980), as parametrized by J. P. Perdew and A. Zunger, Phys. Rev B **23**, 5075 (1981)
- `PW92`: LDA/LSD, as parametrized by J. P. Perdew and Y. Wang, Phys. Rev B, **45**, 13244 (1992)
- `PW91`: Generalized gradients approximation (GGA) of Perdew and Wang. Ref: P&W, J. Chem. Phys., **100**, 1290 (1994)
- `PBE`: GGA of J. P. Perdew, K. Burke and M. Ernzerhof, Phys. Rev. Lett. **77**, 3865 (1996)
- `revPBE`: Modified GGA-PBE functional of Y. Zhang and W. Yang, Phys. Rev. Lett. **80**, 890 (1998)
- `RPBE`: Modified GGA-PBE functional of B. Hammer, L. B. Hansen and J. K. Norskov Phys. Rev. B **59**, 7413 (1999)
- `WC` Modified GGA-PBE functional of Z. Wu and R. E. Cohen, Phys. Rev. B **73**, 235116 (2006)
- `AM05`: Modified GGA-PBE functional of R. Armiento and A. E. Mattsson, Phys. Rev. B **72**, 085108 (2005)
- `PBEsol`: Modified GGA-PBE functional of J. P. Perdew et al, Phys. Rev. Lett. **100**, 136406 (2008)
- `PBEJsJrLO`: GGA-PBE functional with parameters β, μ, and κ fixed by the jellium surface (Js), jellium response (Jr), and Lieb-Oxford bound (LO) criteria, respectively, as described by L. S. Pedroza, A. J. R. da Silva, and K. Capelle, Phys. Rev. B **79**, 201106(R) (2009), and by M. M. Odashima, K. Capelle, and S. B. Trickey, J. Chem. Theory Comput. **5**, 798 (2009)
- `PBEJsJrHEG`: Same as PBEJsJrLO, with parameter κ fixed by the Lieb-Oxford bound for the low density limit of the homogeneous electron gas (HEG)

- `PBEGcGxLO`: Same as PBEJsJrLO, with parameters β and μ fixed by the gradient expansion of correlation (Gc) and exchange (Gx), respectively
- `PBEGcGxHEG`: Same as previous ones, with parameters β, μ, and κ fixed by the Gc, Gx, and HEG criteria, respectively.
- `BLYP` (equivalent to `LYP`): GGA with Becke exchange (A. D. Becke, Phys. Rev. A **38**, 3098 (1988)) and Lee-Yang-Parr correlation (C. Lee, W. Yang, R. G. Parr, Phys. Rev. B **37**, 785 (1988)), as modified by B. Miehlich, A. Savin, H. Stoll, and H. Preuss, Chem. Phys. Lett. **157**, 200 (1989). See also B. G. Johnson, P. M. W. Gill and J. A. Pople, J. Chem. Phys. **98**, 5612 (1993). (Some errors were detected in this last paper, so not all of their expressions correspond exactly to those implemented in SIESTA)
- `DRSLL` (equivalent to `DF1`): van der Waals density functional (vdW-DF) of M. Dion, H. Rydberg, E. Schröder, D. C. Langreth, and B. I. Lundqvist, Phys. Rev. Lett. **92**, 246401 (2004), with the efficient implementation of G. Román-Pérez and J. M. Soler, Phys. Rev. Lett. **103**, 096102 (2009)
- `LMKLL` (equivalent to `DF2`): vdW-DF functional of Dion *et al* (same as DRSLL) reparametrized by K. Lee, E. Murray, L. Kong, B. I. Lundqvist and D. C. Langreth, Phys. Rev. B **82**, 081101 (2010)
- `KBM`: vdW-DF functional of Dion *et al* (same as DRSLL) with exchange modified by J. Klimes, D. R. Bowler, and A. Michaelides, J. Phys.: Condens. Matter **22**, 022201 (2010) (optB88-vdW version)
- `C09`: vdW-DF functional of Dion *et al* (same as DRSLL) with exchange modified by V. R. Cooper, Phys. Rev. B **81**, 161104 (2010)
- `BH`: vdW-DF functional of Dion *et al* (same as DRSLL) with exchange modified by K. Berland and P. Hyldgaard, Phys. Rev. B 89, 035412 (2014)
- `VV`: vdW-DF functional of O. A. Vydrov and T. Van Voorhis, J. Chem. Phys. **133**, 244103 (2010)

Use: **XC.functional** and **XC.authors** must be compatible.

Default value: `PZ`

XC.hybrid (*data block*): This data block allows the user to create a "cocktail" functional by mixing the desired amounts of exchange and correlation from each of the functionals described under XC.authors. Note that these "mixed" functionals do *not* have the exact Hartree-Fock exchange which is a key ingredient of the true "hybrid" functionals. The use of the word "hybrid" in the label is unfortunate in this regard, and might be deprecated in a future version.

The first line of the block must contain the number of functionals to be mixed. On the subsequent lines the values of XC.functl and XC.authors must be given and then the weights for the exchange and correlation, in that order. If only one number is given then the same weight is applied to both exchange and correlation.

The following is an example in which a 75:25 mixture of Ceperley-Alder and PBE correlation is made, with an equal split of the exchange energy:

```
%block XC.hybrid
   2
   LDA CA  0.5 0.75
   GGA PBE 0.5 0.25
%endblock XC.hybrid
```

Default value: `not hybrid`

6.7 Spin polarization

SpinPolarized (*logical*): Logical variable to choose between spin unpolarized (`.false.`) or spin polarized (`.true.`) calculation.

Default value: `.false.`

NonCollinearSpin (*logical*): If `.true.`, non-collinear spin (i.e. with variable orientation) is described using spinor wavefunctions and (2×2) spin density matrices at every grid point. Refs: T. Oda et al, PRL, **80**, 3622 (1998); V. M. García-Suárez et al, Eur. Phys. Jour. B **40**, 371 (2004); V. M. García-Suárez et al, Journal of Phys: Cond. Matt **16**, 5453 (2004). magnitude, and not the gradient of its orientation, are considered to calculate the local xc energy density. Not compatible with the `Diag.ParallelOverK` option *Default value:* `.false.`

FixSpin (*logical*): If `.true.`, the calculation is done with a fixed value of the spin of the system, defined by variable **TotalSpin**. This option can only be used for collinear spin polarized calculations.

Default value: `.false.`

TotalSpin (*real*): Value of the imposed total spin polarization of the system (in units of the electron spin, 1/2). It is only used if **FixSpin** = `.true.`

Default value: 0.0

SingleExcitation (*logical*): If true, SIESTA calculates a very rough approximation to the lowest excited state by swapping the populations of the HOMO and the LUMO. If there is no spin polarisation, it is half swap only. It is done for the first spin component (up) and first k vector.

Default value: `.false.`

6.8 The self-consistent-field loop

Harris_functional (*logical*): Logical variable to choose between self-consistent Kohn-Sham functional or non self-consistent Harris functional to calculate energies and forces.

- `.false.` : Fully self-consistent Kohn-Sham functional.
- `.true.` : Non self consistent Harris functional. Cheap but pretty crude for some systems. The forces are computed within the Harris functional in the first SCF step. Only implemented for LDA in the Perdew-Zunger parametrization.

  When this option is choosen, the values of DM.UseSaveDM, MaxSCFIterations, SCF-MustConverge and DM.MixSCF1 are automatically set up to False, 1, False and False respectively, no matter whatever other specification are in the INPUT file.

Default value: `.false.`

MinSCFIterations (*integer*): Minimum number of SCF iterations per time step. In MD simulations this can with benefit be set to 3.

Default value: `0`

MaxSCFIterations (*integer*): Maximum number of SCF iterations per time step.

Default value: `50`

SCFMustConverge (*logical*): Defines the behaviour if convergence is not reached in the maximum number of SCF iterations. The default is to update the forces, perform an MD or geometry optimisation step and carry on. When set to true the calculation will stop on the first SCF convergence failure.

Default value: `.false.`

6.8.1 Mixing options

MixHamiltonian (*logical*): Mixing of the Hamiltonian instead of the density-matrix, a feature previously available only for TranSiesta runs has been implemented for general use. It is enabled by setting either the **MixHamiltonian** option (preferred) or the old-style **TS.MixH** option (deprecated but retained due to its use in TranSiesta).

The evidence obtained so far from test runs indicates that Hamiltonian mixing might be better than density-matrix mixing in most cases, and is seldom worse. Users are encouraged to test this feature for their favorite tough-converging systems, and report their experiences.

The H mixing algorithms available are exactly the same as for density-matrix mixing (Pulay, Broyden, Fire), and the keywords controlling it are also the same, e.g. `DM.MixingWeight` applies both to H and DM mixing.

DM.MixSCF1 (*logical*): Logical variable to indicate whether mixing is done in the first SCF cycle or not. Usually, mixing should not be done in the first cycle, to avoid non-idempotency in density matrix from Harris or previous steps. It can be useful, though, for restarts of selfconsistency runs.

Default value: `.false.`

SCF.MixAfterConvergence (*logical*): Logical variable to indicate whether mixing is done in the last SCF cycle (after convergence has been achieved) or not. Not mixing after convergence improves the quality of the final Kohn-Sham energy and of the forces when mixing the DM.

Default value: `.false.` **NOTE**: This breaks backwards compatibility with previous (pre-4.0rc) versions of SIESTA. To recover the old behavior, turn this option on, or use the compatibility switch **compat-pre-v4-dm-h** (see below).

SCF.RecomputeHAfterScf (*logical*): Logical variable to indicate whether the Hamiltonian is updated after the scf cycle, while computing the final energy, forces, and stresses. Not recomputing H makes further analysis tasks (such as the computation of band structures) more consistent, as they will be able to use the same H used to generate the last density matrix.

Default value: `.false.` **NOTE**: This breaks backwards compatibility with previous (pre-4.0rc) versions of SIESTA. To recover the old behavior, turn this option on, or use the compatibility switch **compat-pre-v4-dm-h** (see below).

compat-pre-v4-dm-h (*logical*): If this logical variable is turned on, the above two options regarding mixing after convergence and recomputation of H after the scf cycle will be turned on.

DM.MixingWeight (*real*): Proportion α of output Density Matrix to be used for the input Density Matrix of next SCF cycle (linear mixing): $\rho_{in}^{n+1} = \alpha\rho_{out}^{n} + (1-\alpha)\rho_{in}^{n}$.

Default value: `0.25`

Pulay mixing (also known as DIIS extrapolation) is the method of choice for accelerating the convergence of the scf cycle.

DM.NumberPulay (*integer*): It controls the Pulay convergence accelerator. Pulay mixing generally accelerates convergence quite significantly, and can reach convergence in cases where linear mixing cannot. The guess for the $n+1$ iteration is constructed using the input and output matrices of the **DM.NumberPulay** former SCF cycles, in the following way: $\rho_{in}^{n+1} = \alpha_P\bar{\rho}_{out}^{n} + (1-\alpha_P)\bar{\rho}_{in}^{n}$, where $\bar{\rho}_{out}^{n}$ and $\bar{\rho}_{in}^{n}$ are constructed from the previous $N =$**DM.NumberPulay** cycles:

$$\bar{\rho}_{out}^{n} = \sum_{i=1}^{N} \beta_i \rho_{out}^{(n-N+i)} \quad ; \quad \bar{\rho}_{in}^{n} = \sum_{i=1}^{N} \beta_i \rho_{in}^{(n-N+i)}. \tag{3}$$

The values of β_i are obtained by minimizing the distance between $\bar{\rho}_{out}^{n}$ and $\bar{\rho}_{in}^{n}$. The value of α_P is given by default by variable **DM.MixingWeight**, although it can be set directly by **SCF.PulayDamping**.

If **DM.NumberPulay** is 0 or 1, simple linear mixing is performed.

Default value: `0`

SCF.Pulay.Damping (*real*): Proportion α_P of the predicted output Density Matrix to be used for the input Density Matrix of next SCF cycle in Pulay mixing (see above). Typically, this can be significantly higher than the mixing parameter used for linear mixing.

Default value: (the value of **DM.MixingWeight***)*

SCF.PulayMinimumHistory (*integer*):

Pulay mixing might kick in only after a specified number of history steps have been built up.

Default value: `2`

SCF.PulayDmaxRegion (*real*):

Pulay mixing might not work well if far from the fixed point. This option will avoid inserting the current X_{in}, X_{out} pair (X could be the density matrix or the hamiltonian) in the history stack if the maximum difference is above the specified number.

Default value: (a very high number, so no effect by default)

DM.NumberKick (*integer*): Option to skip the Pulay (or Broyden) mixing each certain number of iterations, and use a linear mixing instead. Linear mixing is done every **DM.NumberKick** iterations, using a mixing coefficient α given by variable **DM.KickMixingWeight** (instead of the usual mixing **DM.MixingWeight**). This allows in some difficult cases to bring the SCF out of loops in which the selfconsistency is stuck. If **DM.MixingWeight**=0, no linear mix is used.

Default value: `0`

DM.KickMixingWeight (*real*): Proportion α of output Density Matrix to be used for the input Density Matrix of next SCF cycle (linear mixing): $\rho_{in}^{n+1} = \alpha\rho_{out}^{n} + (1-\alpha)\rho_{in}^{n}$, for linear mixing kicks within the Pulay or Broyden mixing schemes. This mixing is done every **DM.NumberKick** cycles.

Default value: `0.50`

DM.Pulay.Avoid.First.After.Kick (*logical*): Controls whether the first density-matrix residual of the SCF cycle and the first residual after a kick are included in the Pulay history. It can be argued that in these cases the "output" DM might be significantly different from the "input" DM. To preserve backwards compatibility, these residuals are kept in the Pulay history unless this variable is activated.

Default value: `.false.`

SCF.LinearMixingAfterPulay (*logical*):

The damping of the DIIS-predicted X_{in} is done to avoid introducing linear dependencies into the Pulay history stack. Alternatively (or simultaneously) one can use the most recent X_{in}, X_{out} pair in a linear mixing step, and use a possibly different mixing parameter. This would be akin to a "'kick", but without removing all the history information.

Default value: `.false.`

SCF.MixingWeightAfterPulay (*real*):

Proportion α of output Density Matrix to be used for the input Density Matrix of next SCF cycle, for linear mixing steps after a Pulay mixing step if the option **SCF.LinearMixingAfterPulay** is activated.

Default value: `0.50`

SCF.Pulay.UseSVD (*logical*):

Instead of a direct matrix inversion, the more robust SVD algorithm can be used to perform the DIIS extrapolation.

Default value: `.false.`

SCF.Pulay.DebugSVD (*logical*):

Print more information (effective rank, singular values) if the SVD algorithm isused to perform the DIIS extrapolation.

Default value: `.true.` *when using SVD*

SCF.Pulay.RcondSVD (*real*):

Singular values which are smaller than `rcond` times the maximum singular value are effectively discarded in the SVD algorithm for solving the DIIS equations. This lowers the effective rank of the problem, and is a sign of (near) linear dependencies in the extrapolation data.

Default value: 10^{-8}

DM.PulayOnFile (*logical*):

NOTE: This feature is temporarily disabled pending a proper implementation that works well in parallel.

Store intermediate information of Pulay mixing in files (`.true.`) or in memory (`.false.`). Memory storage can increase considerably the memory requirements for large systems. If files are used, the filenames will be `SystemLabel`.P1 and `SystemLabel`.P2, where SystemLabel is the name associated to parameter `SystemLabel`.

Default value: `.false.`

DM.NumberBroyden (*integer*): It controls the Broyden-Vanderbilt-Louie-Johnson convergence accelerator, which is based on the use of past information (up to **DM.NumberBroyden** steps) to construct the input density matrix for the next iteration.

See D.D. Johnson, Phys. Rev. B**38**, 12807 (1988), and references therein; Kresse and Furthmuller, Comp. Mat. Sci **6**, 15 (1996).

If **DM.NumberBroyden** is 0, the program performs linear mixings, or, if requested, Pulay mixings.

Broyden mixing takes precedence over Pulay mixing if both are specified in the input file.

Note: The Broyden mixing algorithm is still in development, notably with regard to the effect of its various modes of operation, and the assigment of weights. In its default mode, its effectiveness is very similar to Pulay mixing. As memory usage is not yet optimized, casual users might want to stick with Pulay mixing for now.

Default value: `0`

DM.Broyden.Cycle.On.Maxit (*logical*): Upon reaching the maximum number of historical data sets which are kept for Broyden mixing (see description of variable

DM.NumberBroyden), throw away the oldest and shift the rest to make room for a new data set. This procedure tends, heuristically, to perform better than the alternative, which is to re-start the Broyden mixing algorithm from a first step of linear mixing.

Default value: `.true.`

DM.Broyden.Variable.Weight (*logical*): If `.true.`, the different historical data sets used in the Broyden mixing (see description of variable **DM.NumberBroyden**) are assigned a weight depending on the norm of their residual $\rho^n_{out} - \rho^n_{in}$.

Default value: `.true.`

6.8.2 Mixing of the Charge Density

MixCharge (*logical*):

If set, this enables an experimental feature, in which the fourier components of the charge density are mixed, as done in some plane-wave codes. (See for example Kresse and Furthmüller, Comp. Mat. Sci. 6, 15-50 (1996), KF in what follows.) Default: `.false.`

The charge mixing is implemented roughly as follows:

- The charge density computed in dhscf is fourier-transformed and stored in a new module. This is done both for "$\rho(\mathbf{G})$(in)" and "$\rho(\mathbf{G})$(out)" (the "out" charge is computed during the extra call to dhscf for correction of the variational character of the Kohn-Sham energy)
- The "in" and "out" charges are mixed (see below), and the resulting "in" fourier components are used by dhscf in successive iterations to reconstruct the charge density.
- The new arrays needed and the processing of most new options is done in the new module m_rhog.F90. The fourier-transforms are carried out by code in rhofft.F.
- Following standard practice, two options for mixing are offered:
  - A simple Kerker mixing, with an optional Thomas-Fermi wavevector to damp the contributions for small G's. The overall mixing weight is the same as for other kinds of mixing, read from **DM.MixingWeight**.
  - A DIIS (Pulay) procedure that takes into account a sub-set of the G vectors (those within a smaller cutoff). Optionally, the scalar product used for the construction of the DIIS matrix from the residuals uses a weight factor.

    The DIIS extrapolation is followed by a Kerker mixing step.

    The code is m_diis.F90. The DIIS history is kept in a circular stack, implemented using the new framework for reference-counted types. This might be overkill for this particular use, and there are a few rough edges, but it works well.

The default convergence criteria remains based on the differences in the density matrix, but in this case the differences are from step to step, not the more fundamental `DM_out-DM_in`. Perhaps some other criterion should be made the default (max —Delta rho(G)—, convergence of the free-energy...)

Note that with charge mixing the Harris energy as it is currently computed in Siesta loses its meaning, since there is no `DM_in`. The program prints zeroes in the Harris energy field.

Note that the KS energy is correctly computed throughout the scf cycle, as there is an extra step for the calculation of the charge stemming from `DM_out`, which also updates the energies. Forces and final energies are correctly computed with the final `DM_out`, regardless of the setting of the option for mixing after scf convergence.

Initial tests suggest that charge mixing has some desirable properties and could be a drop-in replacement for density-matrix mixing, but many more tests are needed to calibrate its efficiency for different kinds of systems, and the heuristics for the (perhaps too many) parameters:

SCF.Kerker.q0sq (*physical energy*):

Determines the parameter q_0^2 featuring in the Kerker preconditioning, which is always performed on all components of $\rho(\mathbf{G})$, even those treated with the DIIS scheme.

Default value: 0.0 Ry.

SCF.RhoGMixingCutoff (*physical energy*):

Determines the sub-set of G vectors which will undergo the DIIS procedure. Only those with kinetic energies below this cutoff will be considered. The optimal extrapolation of the $\rho(\mathbf{G})$ elements will be replaced in the fourier series before performing the Kerker mixing.

Default value: 9.0 Ry.

SCF.RhoG.DIIS.Depth (*integer*):

Determines the maximum number of previous steps considered in the DIIS procedure.

Default value: 0 (no DIIS procedure performed).

NOTE: The information from the first scf step is not included in the DIIS history. There is no provision yet for any other kind of "kick-starting" procedure. The logic is in m_rhog (rhog_mixing routine).

SCF.RhoG.Metric.Preconditioner.Cutoff (*physical energy*):

Determines the value of q_1^2 in the weighing of the different $\mathbf{G}$ components in the scalar products among residuals in the DIIS procedure. Following the KF ansatz, this parameter is chosen so that the smallest (non-zero) $\mathbf{G}$ has a weight 20 times larger than that of the smallest G vector in the DIIS set.

Default value: The result of the KF prescription.

SCF.DebugRhogMixing (*logical*):

Controls the level of debugging output in the mixing procedure (basically whether the first few stars worth of Fourier components are printed). Note that this feature will only display the components in the master node.

Default: `.false.`

DebugDIIS (*logical*):

Controls the level of debugging output in the DIIS procedure. If set, the program prints the DIIS matrix and the extrapolation coefficients.

Default: `.false.`

SCF.MixCharge.SCF1 (*logical*):

Logical variable to indicate whether or not the charge is mixed in the first SCF cycle. Anecdotal evidence indicates that it might be advantageous, at least for calculations started from scratch, to avoid that first mixing, and retain the "out" charge density as "in" for the next step.

Default: `.false.`

6.8.3 Initialization of the density-matrix

The Density matrix can be:

```
1. Synthesized directly from atomic occupations.
   (See the options below for spin considerations)
2. Read from a .DM file (if the appropriate options are set)
3. Extrapolated from (two) previous geometry steps
3.a The DM of the previous geometry iteration

In cases 2 and 3, a check is done to guarantee that the structure
of the read or extrapolated DM conforms to the current sparsity.
If it does not, the information is re-arranged.

Special cases:
       Harris functional: The matrix is always initialized
       Force calculation: The DM should be written to disk
                          at the time of the "no displacement"
                          calculation and read from file at
                          every subsequent step.
       Variable-cell calculation:
         If the auxiliary cell changes, the DM is forced to be
         initialized (conceivably one could rescue some important
         information from an old DM, but it is too much trouble
         for now). NOTE that this is a change in policy with respect
         to previous versions of the program, in which a (blind?)
         re-use was allowed, except if 'ReInitialiseDM' was 'true'.
         Now 'ReInitialiseDM' is 'true' by default. Setting it to
         'false' is not recommended.

         In all other cases (including "server operation"), the
         default is to allow DM re-use (with possible extrapolation)
         from previous geometry steps.
```

```
There is no re-use of the DM for "Forces", and "Phonon"
dynamics types (i.e., the DM is re-initialized)

For "CG" calculations, the default is not to extrapolate the
DM (unless requested by setting 'DM.AllowExtrapolation' to
"true"). The previous step's DM is reused.

The fdf variables 'DM.AllowReuse' and 'DM.AllowExtrapolation'
can be used to turn off DM re-use and extrapolation.
```

DM.UseSaveDM (*logical*): Instructs to read the density matrix stored in file `SystemLabel`.DM by a previous run.

Use: If the required file does not exist, a warning is printed but the program does not stop. Overrides **UseSaveData**.

Default value: `.false.`

DM.FormattedFiles (*logical*): Instructs to use formatted files for reading and writing the density matrix. In this case, the files are labelled `SystemLabel`.DMF.

Use: This makes for much larger files, and slower i/o. However, the files are transferable between different computers, which is not the case normally.

Default value: `.false.`

DM.FormattedInput (*logical*): Instructs to use formatted files for reading the density matrix.

Use: Overrides the value of **DM.FormattedFiles**.

Default value: `.false.`

DM.FormattedOutput (*logical*): Instructs to use formatted files for writing the density matrix.

Use: Overrides the value of **DM.FormattedFiles**.

Default value: `.false.`

DM.InitSpinAF (*logical*): It defines the initial spin density for a spin polarized calculation. The spin density is initially constructed with the maximum possible spin polarization for each atom in its atomic configuration. This variable defines the relative orientation of the atomic spins:

- `.false.` gives ferromagnetic order (all spins up).
- `.true.` gives antiferromagnetic order. Up and down are assigned according to order in the block **AtomicCoordinatesAndAtomicSpecies**: up for the odd atoms, down for even.

Default value: `.false.`

DM.InitSpin (*data block*): It defines the initial spin density for a spin polarized calculation atom by atom. In the block there is one line per atom to be spin-polarized, containing the atom index (integer, ordinal in the block **AtomicCoordinatesAndAtomicSpecies**) and the desired initial spin-polarization (real, positive for spin up, negative for spin down). A value larger than possible will be reduced to the maximum possible polarization, keeping its sign. Maximum polarization can also be given by introducing the symbol `+` or `-` instead of the polarization value. There is no need to include a line for every atom, only for those to be polarized. The atoms not contemplated in the block will be given non-polarized initialization. For non-collinear spin, the spin direction may be specified for each atom by the polar angles theta and phi, given as the last two arguments in degrees. If not specified, theta=0 is assumed. **NonCollinearSpin** must be `.true.` to use the spin direction.

Example:

```
%block DM.InitSpin
   5  -1.   90.   0.   # Atom index, spin, theta, phi (deg)
   3   +    45. -90.
   7   -
%endblock DM.InitSpin
```

Default value: If present but empty, all atoms are not polarized. If absent, **DM.InitSpinAF** defines the polarization.

DM.AllowReuse (*logical*): Controls whether density matrix information from previous geometry iterations is re-used to start the new geometry's SCF cycle.

Default value: `.true.`

DM.AllowExtrapolation (*logical*): Controls whether density matrix information from two previous geometry iterations is (linearly) extrapolated to start the new geometry's SCF cycle.

Default value: `.true.`

Further information regarding the density-matrix re-use can be found in the header of routine `Src/new_dm.F`.

6.8.4 Initialization of the SCF cycle with charge densities

SCF.Read.Charge.NetCDF (*logical*): Instructs Siesta to read the charge density stored in the netCDF file `Rho.IN.grid.nc`. This feature allows the easier re-use of electronic-structure information from a previous run. It is not necessary that the basis sets are "similar" (a requirement if density-matrices are to be read in).

Use: This is an experimental feature. Until robust checks are implemented, care must be taken to make sure that the FFT grids in the .grid.nc file and in Siesta are the same.

Default value: `.false.`

SCF.Read.Deformation.Charge.NetCDF (*logical*): Instructs Siesta to read the deformation charge density stored in the netCDF file `DeltaRho.IN.grid.nc`. This feature allows the easier re-use of electronic-structure information from a previous run. It is not necessary that the basis sets are "similar" (a requirement if density-matrices are to be read in). The deformation charge is particularly useful to give a good starting point for slightly different geometries.

Use: This is an experimental feature. Until robust checks are implemented, care must be taken to make sure that the FFT grids in the .grid.nc file and in Siesta are the same.

Default value: `.false.`

6.8.5 Output of density matrix

Performance Note: For large-scale calculations, writing the DM at every scf step can have a severe impact on performance.

WriteDM (*logical*): It determines whether the density matrix is output as a binary *Systemlabel*.DM file or not.

Default value: `.true.`

WriteDM.NetCDF (*logical*): It determines whether the density matrix (after the mixing step) is output as a DM.nc netCDF file or not.

The file is overwritten at every SCF step. Use the **WriteDM.History.NetCDF** option if a complete history is desired.

The DM.nc and standard DM file formats can be converted at will with the programs in `Util/DensityMatrix` directory. Note that the DM values in the DM.nc file are in single precision.

Default value: `.true.` if netCDF support is enabled (see Appendix).

WriteDMHS.NetCDF (*logical*): If true, the input density matrix, Hamiltonian, and output density matrix, are stored in a netCDF file named `DMHS.nc`. The file also contains the overlap matrix S.

The file is overwritten at every SCF step. Use the **WriteDMHS.History.NetCDF** option if a complete history is desired.

Default value: `.true.` if netCDF support is enabled (see Appendix).

WriteDM.History.NetCDF (*logical*): If true, a series of netCDF files with names of the form `DM-NNNN.nc` is created to hold the complete history of the density matrix (after mixing). (See also **WriteDM.NetCDF**). Each file corresponds to a geometry step.

Default value: `.false.`

WriteDMHS.History.NetCDF (*logical*): If true, a series of netCDF files with names of the form `DMHS-NNNN.nc` is created to hold the complete history of the input and output density matrix, and the Hamiltonian. (See also **WriteDMHS.NetCDF**). Each file corresponds to a geometry step. The overlap matrix is stored only once per SCF cycle.

Default value: `.false.`

6.8.6 Convergence criteria

NOTE: Some of the options below have a `DM` prefix. This is for historical reasons, as the density-matrix convergence was typically the only criterion for termination of the scf cycle.

DM.Tolerance (*real*): Tolerance of Density Matrix. When the maximum difference between the output and the input on each element of the DM in a SCF cycle is smaller than DM.Tolerance, the selfconsistency has been achieved.

Default value: 10^{-4}

DM.Require.Energy.Convergence (*logical*): Logical variable to request an additional requirement for self-consistency: it is considered achieved when the change in the total (free) energy between cycles of the SCF procedure is below **DM.EnergyTolerance** and the density matrix change criterion is also satisfied.

Default value: `.false.`

DM.Energy.Tolerance (*real energy*): If **DM.Require.Energy.Convergence** is `.true.`, then self-consistency is achieved when the change in the total (free) energy between cycles of the SCF procedure is below this value and the density matrix change criterion is also satisfied.

Default value: 10^{-4} eV

DM.Require.Harris.Convergence (*logical*): Logical variable to use the Harris energy as monitor of self-consistency: this is considered achieved when the change in the Harris energy between cycles of the SCF procedure is below **DM.Harris.Tolerance**. No density matrix change criterion is used. This is useful if only energies are needed, as the Harris energy tends to converge faster than the Kohn-Sham energy. The user is responsible for using the correct energies in further processing, e.g., the Harris energy if the Harris criterion is used.

To help in basis-optimization tasks, a new file BASIS_HARRIS_ENTHALPY is provided, holding the same information as BASIS_ENTHALPY but using the Harris energy instead of the Kohn-Sham energy.

Default value: `.false.`

DM.Harris.Tolerance (*real energy*): If **DM.Require.Harris.Convergence** is `.true.`, then self-consistency is achieved when the change in the Harris energy between cycles of the SCF procedure is below this value. This is useful if only energies are needed, as the Harris energy tends to converge faster than the Kohn-Sham energy.

Default value: 10^{-4} eV

6.9 The real-space grid and the eggbox-effect

SIESTA uses a finite 3D grid for the calculation of some integrals and the representation of charge densities and potentials. Its fineness is determined by its plane-wave cutoff, as given by

the **MeshCutoff** option. It means that all periodic plane waves with kinetic energy lower than this cutoff can be represented in the grid without aliasing. In turn, this implies that if a function (e.g. the density or the effective potential) is an expansion of only these plane waves, it can be Fourier transformed back and forth without any approximation.

The existence of the grid causes the breaking of translational symmetry (the egg-box effect, due to the fact that the density and potential *do have* plane wave components above the mesh cutoff). This symmetry breaking is clear when moving one single atom in an otherwise empty simulation cell. The total energy and the forces oscillate with the grid periodicity when the atom is moved, as if the atom were moving on an eggbox. In the limit of infinitely fine grid (infinite mesh cutoff) this effect disappears.

For reasonable values of the mesh cutoff, the effect of the eggbox on the total energy or on the relaxed structure is normally unimportant. However, it can affect substantially the process of relaxation, by increasing the number of steps considerably, and can also spoil the calculation of vibrations, usually much more demanding than relaxations.

The Util/Scripting/eggbox_checker.py script can be used to diagnose the eggbox effect to be expected for a particular pseudopotential/basis-set combination.

Apart from increasing the mesh cutoff (see the **MeshCutoff** option), the following options might help in lessening a given eggbox problem. But note also that a filtering of the orbitals and the relevant parts of the pseudopotential and the pseudocore charge might be enough to solve the issue (see Sect. 6.3.9).

MeshCutoff (*real energy*): Defines the plane wave cutoff for the grid.

Default value: `100 Ry`

MeshSubDivisions (*integer*): Defines the number of sub-mesh points in each direction used to save index storage on the mesh. It affects the memory requirements and the CPU time, but not the results.

Default value: `2`

NOTE: The default value might be a bit conservative. Users might experiment with higher values (4, 6) to lower the memory and cputime usage.

GridCellSampling (*data block/list*):

It specifies points within the grid cell for a symmetrization sampling.

For a given grid the grid-cutoff convergence can be improved (and the eggbox lessened) by recovering the lost symmetry: by symmetrizing the sensitive quantities. The full symmetrization implies an integration (averaging) over the grid cell. Instead, a finite sampling can be performed.

It is a sampling of rigid displacements of the system with respect to the grid. The original grid-system setup (one point of the grid at the origin) is always calculated. It is the (0,0,0) displacement. The block **GridCellSampling** gives the additional displacements wanted for the sampling. They are given relative to the grid-cell vectors, i.e., (1,1,1) would displace to the next grid point across the body diagonal, giving an equivalent grid-system situation (a useless displacement for a sampling).

Examples: Assume a cubic cell, and therefore a (smaller) cubic grid cell. If there is no block or the block is empty, then the original (0,0,0) will be used only. The block:

```
%block GridCellSampling
   0.5    0.5    0.5
%endblock GridCellSampling
```

would use the body center as a second point in the sampling. Or:

```
%block GridCellSampling
   0.5    0.5    0.0
   0.5    0.0    0.5
   0.0    0.5    0.5
%endblock GridCellSampling
```

gives an fcc kind of sampling, and

```
%block GridCellSampling
   0.5    0.0    0.0
   0.0    0.5    0.0
   0.0    0.0    0.5
   0.0    0.5    0.5
   0.5    0.0    0.5
   0.5    0.5    0.0
   0.5    0.5    0.5
%endblock GridCellSampling
```

gives again a cubic sampling with half the original side length. It is not trivial to choose a right set of displacements so as to maximize the new 'effective' cutoff. It depends on the kind of cell. It may be automatized in the future, but it is now left to the user, who introduces the displacements manually through this block.

The quantities which are symmetrized are: (*i*) energy terms that depend on the grid, (*ii*) forces, (*iii*) stress tensor, and (*iv*) electric dipole.

The symmetrization is performed at the end of every SCF cycle. The whole cycle is done for the (0,0,0) displacement, and, when the density matrix is converged, the same (now fixed) density matrix is used to obtain the desired quantities at the other displacements (the density matrix itself is *not* symmetrized as it gives a much smaller egg-box effect). The CPU time needed for each displacement in the **GridCellSampling** block is of the order of one extra SCF iteration.

This may be required in systems where very precise forces are needed, and/or if partial cores are used. It is advantageous to test whether the forces are sampled sufficiently by sampling one point.

Default value: Empty.

EggboxRemove (*data block*):

For recovering translational invariance in an approximate way.

It works by substracting from Kohn-Sham's total energy (and forces) an approximation to the eggbox energy, sum of atomic contributions. Each atom has a predefined eggbox energy depending on where it sits on the cell. This atomic contribution is species dependent and is obviously invariant under grid-cell translations. Each species contribution is thus expanded in the appropriate Fourier series. It is important to have a smooth eggbox, for it to be represented by a few Fourier components. A jagged egg-box (unless very small, which is then unimportant) is often an indication of a problem with the pseudo.

In the block there is one line per Fourier component. The first integer is for the atomic species it is associated with. The other three represent the reciprocal lattice vector of the grid cell (in units of the basis vectors of the reciprocal cell). The real number is the Fourier coefficient in units of the energy scale given in **EggboxScale** (see below), normally 1 eV.

The number and choice of Fourier components is free, as well as their order in the block. One can choose to correct only some species and not others if, for instance, there is a substantial difference in hardness of the cores. The 0 0 0 components will add a species-dependent constant energy per atom. It is thus irrelevant except if comparing total energies of different calculations, in which case they have to be considered with care (for instance by putting them all to zero, i.e. by not introducing them in the list). The other components average to zero representing no bias in the total energy comparisons.

If the total energies of the free atoms are put as 0 0 0 coefficients (with spin polarisation if adequate etc.) the corrected total energy will be the cohesive energy of the system (per unit cell).

Example: For a two species system, this example would give a quite sufficent set in many instances (the actual values of the Fourier coefficients are not realistic).

```
%block EggBoxRemove
  1   0   0   0 -143.86904
  1   0   0   1    0.00031
  1   0   1   0    0.00016
  1   0   1   1   -0.00015
  1   1   0   0    0.00035
  1   1   0   1   -0.00017
  2   0   0   0 -270.81903
  2   0   0   1    0.00015
  2   0   1   0    0.00024
  2   1   0   0    0.00035
  2   1   0   1   -0.00077
  2   1   1   0   -0.00075
  2   1   1   1   -0.00002
%endblock EggBoxRemove
```

It represents an alternative to grid-cell sampling (above). It is only approximate, but once the Fourier components for each species are given, it does not represent any computational

effort (neither memory nor time), while the grid-cell sampling requires CPU time (roughly one extra SCF step per point every MD step).

It will be particularly helpful in atoms with substantial partial core or semicore electrons.

Use: This technique as it stands should only be used for fixed cell calculations.

For the time being, it is up to the user to obtain the Fourier components to be introduced. They can be obtained by moving one isolated atom through the cell to be used in the calculation (for a give cell size, shape and mesh), once for each species. The Util/Scripting/eggbox_checker.py script can be used as a starting point for this.

Default value: Empty.

EggboxScale (*real energy*):

Defines the scale in which the Fourier components of the egg-box energy are given in the **EggboxRemove** block.

Default value: 1 eV.

6.10 Matrix elements of the Hamiltonian and overlap

NeglNonOverlapInt (*logical*): Logical variable to neglect or compute interactions between orbitals which do not overlap. These come from the KB projectors. Neglecting them makes the Hamiltonian more sparse, and the calculation faster. **Use with care**

Default value: `.false.`

SaveHS (*logical*): Instructs to write the Hamiltonian and overlap matrices, as well as other data required to generate bands and density of states, in file `SystemLabel`.HSX. The HSX format is more compact than the traditional HS, and the Hamiltonian, overlap matrix, and relative-positions array (which is always output, even for gamma-point only calculations) are in single precision.

The program `hsx2hs` in `Util/HSX` can be used to generate an old-style HS file if needed.

SIESTA produces also an HSX file if the **COOP.Write** option is active.

Use: File `SystemLabel`.HS is only written, not read, by siesta.

Default value: `.false.`

See also the **WriteDMHS.NetCDF** and **WriteDMHS.History.NetCDF** options.

Compat.Matel.NRTAB (*logical*):

Internally the two-center integrals involved in some matrix element calculations are tabulated with a preset number of elements. In versions 4.0.1 and prior this was 128. Since 4.0.2 the number of table elements has been increased to 1024, which translates to more accurate matrix element calculations.

This compatibility option should *only* be used when preservation of the (lower accuracy) numerical results of 4.0.1 or prior versions is required for reproducibility purposes.

Default value: `.false.`

6.10.1 The auxiliary supercell

When using k-points, this auxiliary supercell is needed to compute properly the matrix elements involving orbitals in different unit cells. It is computed automatically by the program at every geometry step.

Note that for gamma-point-only calculations there is an implicit "folding" of matrix elements corresponding to the images of orbitals outside the unit cell. If information about the specific values of these matrix elements is needed (as for COOP/COHP analysis), one has to make sure that the unit cell is large enough, or force the use of an aunxiliary supercell.

FixAuxiliaryCell (*logical*):

Logical variable to control whether the auxiliary cell is changed during a variable cell optimization. If this option is used, care should be taken that the supercell is appropriate for the whole duration of the run.

Default value: `.false.`

NaiveAuxiliaryCell (*logical*):

If true, the program does not check whether the auxiliary cell constructed with a naive algorithm is appropriate. This variable should only be used if one wishes to reproduce calculations done with previous versions of the program in which the auxiliary cell was not large enough, as indicated by warnings such as:

```
WARNING: orbital pair 1 341 is multiply connected
```

Only small numerical differences in the results are to be expected.

Default value: `.false.`

ForceAuxCell (*logical*):

If true, the program uses an auxiliary cell even for gamma-point-only calculations. This might be needed for COOP/COHP calculations, as noted above, or in degenerate cases, such as when the cell is so small that a given orbital "self-interacts" with its own images (via direct overlap or through a KB projector). In this case, the diagonal value of the overlap matrix S for this orbital is different from 1, and an initialization of the DM via atomic data would be faulty. The program corrects the problem to zeroth-order by dividing the DM value by the corresponding overlap matrix entry, but the initial charge density would exhibit distortions from a true atomic superposition (See routine `new_dm.F`). The distortion of the charge density is a serious problem for Harris functional calculations, so this option must be enabled for them if self-folding is present. (Note that this should not happen in any serious calculation...)

Default value: `.false.`

6.11 Calculation of the electronic structure

SIESTA can use two methods to determine the electronic structure of the system. One is standard diagonalization, which works for all systems and has a cubic scaling with the size. The other

is based on the direct minimization of a special functional over a set of trial orbitals. These orbitals can either extend over the entire system, resulting in a cubic scaling algorithm, or be constrained within a localization radius, resulting in a linear scaling algorithm. The former is a very recent implementation (described in 6.11.4), that can be viewed as an equivalent approach to diagonlization in terms of the accuracy of the solution; the latter is the historic O(N) method used by SIESTA (described in 6.11.5); it scales in principle linearly with the size of the system (only if the size is larger than the radial cutoff for the local solution wave-functions), but is quite fragile and substantially more difficult to use, and only works for systems with clearly separated occupied and empty states. The default is to use diagonalization.

The calculation of the H and S matrix elements is always done with an O(N) method. The actual scaling is not linear for small systems, but it becomes O(N) when the system dimensions are larger than the scale of orbital r_c's.

The relative importance of both parts of the computation (matrix elements and solution) depends on the size and quality of the calculation. The mesh cutoff affects only the matrix-element calculation; orbital cutoff radii affect the matrix elements and the linear-scaling solver, but not the diagonalization; the need for **k**-point sampling affects the solvers only, and the number of basis orbitals affects them all.

In practice, the vast majority of users employ diagonalization for the calculation of the electronic structure. This is so because the vast majority of calculations would not benefit from the O(N) solver (most of the calculations done are for intermediate system sizes, for varied reasons e.g. the long time scales needed in MD simulations for growing system sizes).

Since it is used less often, bugs creeping into the O(N) solver have been more resilient than in more popular bits of the code. Work is ongoing to clean and automate the O(N) process, to make the solver more user-friendly and robust.

SolutionMethod (*string*): Character string to chose between diagonalization (`diagon`), cubic-scaling minimization (`OMM`) or Order-N (`OrderN`) solution of the Kohn-Sham Hamiltonian.

Default value: `diagon`

6.11.1 Diagonalization options

NumberOfEigenStates (*integer*): This parameter allows the user to reduce the number of eigenstates that are calculated from the maximum possible. The benefit is that, for a gamma point calculation, the cost of the diagonalisation is reduced by finding fewer eigenvectors. For example, during a geometry optimisation, only the occupied states are required rather than the full set of virtual orbitals. Note, that if the electronic temperature is greater than zero then the number of partially occupied states increases, depending on the band gap. The value specified must greater than the number of occupied states and less than the number of basis functions.

Default value: `all orbitals`

Use.New.Diagk (*logical*): Selects whether a more efficient diagonalization routine (with intermediate storage of eigenvectors in netCDF format) is used for the case of k-point sampling.

In order to use the new routine, netCDF support should be compiled in. Specifying a number of eigenvectors to store is possible through the symbol NumberOfEigenstates (see above). Note that for now, for safety, all eigenvectors for a given k-point and spin are computed by the diagonalization routine, but only that number specified by the user are stored. If they are insufficient, the program stops. A rule of thumb to select the number of eigenvectors to store is to count the number of electrons and divide by two, and then apply a "safety factor" of around 1.1-1.2 to take into account fractional occupations and band overlaps.

A new file `OCCS` is produced with information about the number of states occupied.

This is an experimental feature. Note: It is not compatible with the `Diag.Parallel.Over.K` option.

Default value: `.false.`

Diag.DivideAndConquer (*logical*): Logical to select whether the normal or Divide and Conquer algorithms are used within the Lapack diagonalisation routines.

(Note: Some system library implementations of the D&C algorithm are buggy. It is advisable to use Siesta's own (fixed) version – configure will try to do that.)

Default value: `true`

Diag.AllInOne (*logical*): Logical to select whether a single call to lapack/scalapack is made to perform the diagonalisation or whether the individual steps are controlled by Siesta. Normally this option should not need to be used.

Default value: `false`

Diag.NoExpert (*logical*): Logical to select whether the simple or expert versions of the lapack/scalapack routines are used. Usually the expert routines are faster, but may require slightly more memory.

Default value: `false`

Diag.PreRotate (*logical*): Logical to select whether the eigensystem is transformed according to previously saved eigenvectors to create a near diagonal matrix and then back transformed afterwards. This is included for future options, but currently should not make any difference except to increase the computational work!

Default value: `false`

Diag.Use2D (*logical*): Logical to select whether a 1-D or 2-D data decomposition should be used when calling scalapack. The use of 2-D leads to superior scaling to large numbers of processors and is therefore the default. This option only influences the parallel performance.

Default value: `true`

6.11.2 Output of eigenvalues and wavefunctions

This section focuses on the output of eigenvalues and wavefunctions produced during the (last) iteration of the self-consistent cycle, and associated to the appropriate k-point sampling.

For band-structure calculations (which typically use a different set of k-points) and specific requests for wavefunctions, see Secs. 6.12 and 6.13, respectively.

The complete set of wavefunctions obtained during the last iteration of the SCF loop will be written to a NetCDF file `WFS.nc` if the **Diag.UseNewDiagk** option is in effect.

The complete set of wavefunctions obtained during the last iteration of the SCF loop will be written to SystemLabel.fullBZ.WFSX if the **COOP.write** option is in effect.

WriteEigenvalues (*logical*): If `.true.` it writes the Hamiltonian eigenvalues for the sampling $\vec{k}$ points, in the main output file. If `.false.`, it writes them in the file *Systemlabel*.EIG, which can be used by the EIG2DOS postprocessing utility (in the Util/Eig2DOS directory) for obtaining the density of states.

Use: Only if **SolutionMethod** is `diagon`.

Default value: `.false.` (see **LongOutput**)

6.11.3 Occupation of electronic states and Fermi level

OccupationFunction (*string*): String variable to select the function that determines the occupation of the electronic states. Two options are available:

- `FD`: The usual Fermi-Dirac occupation function is used.
- `MP`: The occupation function proposed by Methfessel and Paxton (Phys. Rev. B, **40**, 3616 (1989)), is used.

The smearing of the electronic occupations is done, in both cases, using an energy width defined by the **ElectronicTemperature** variable. Note that, while in the case of Fermi-Dirac, the occupations correspond to the physical ones if the electronic temperature is set to the physical temperature of the system, this is not the case in the Methfessel-Paxton function. In this case, the tempeature is just a mathematical artifact to obtain a more accurate integration of the physical quantities at a lower cost. In particular, the Methfessel-Paxton scheme has the advantage that, even for quite large smearing temperatures, the obtained energy is very close to the physical energy at T=0. Also, it allows a much faster convergence with respect to k-points, specially for metals. Finally, the convergence to selfconsistency is very much improved (allowing the use of larger mixing coefficients).

For the Methfessel-Paxton case, one can use relatively large values for the **ElectronicTemperature** parameter. How large depends on the specific system. A guide can be found in the article by J. Kresse and J. Furthmüller, Comp. Mat. Sci. **6**, 15 (1996).

If Methfessel-Paxton smearing is used, the order of the corresponding Hermite polynomial expansion must also be chosen (see description of variable **OccupationMPOrder**).

We finally note that, in both cases (FD and MP), once a finite temperature has been chosen, the relevant energy is not the Kohn-Sham energy, but the Free energy. In particular, the

atomic forces are derivatives of the Free energy, not the KS energy. See R. Wentzcovitch *et al.*, Phys. Rev. B **45**, 11372 (1992); S. de Gironcoli, Phys. Rev. B **51**, 6773 (1995); J. Kresse and J. Furthmüller, Comp. Mat. Sci. **6**, 15 (1996), for details.

Use: Used only if **SolutionMethod** = `diagon`

Default value: `FD`

OccupationMPOrder (*integer*): Order of the Hermite-Gauss polynomial expansion for the electronic occupation functions in the Methfessel-Paxton scheme (see Phys. Rev. B **40**, 3616 (1989)). Specially for metals, higher order expansions provide better convergence to the ground state result, even with larger smearing temperatures, and provide also better convergence with k-points.

Use: Used only if **SolutionMethod** = `diagon` and **OccupationFunction** = `MP`

Default value: `1`

ElectronicTemperature (*real temperature or energy*): Temperature for Fermi-Dirac or Methfessel-Paxton distribution. Useful specially for metals, and to accelerate selfconsistency in some cases.

Use: Used only if **SolutionMethod** = `diagon`

Default value: `300.0 K` pp

6.11.4 Orbital minimization method (OMM)

The OMM is an alternative cubic-scaling solver that uses a minimization algorithm instead of direct diagonalization to find the occupied subspace. The main advantage over diagonalization is the possibility of iteratively reusing the solution from each SCF/MD step as the starting guess of the following one, thus greatly reducing the time to solution. Typically, therefore, the first few SCF cycles of the first MD step of a simulation will be slower than diagonalization, but the rest will be faster. The main disadvantages are that individual Kohn-Sham eigenvalues are not computed, and that only a fixed, integer number of electrons at each k point/spin is allowed. Therefore, only spin-polarized calculations with **FixSpin** are allowed, and **TotalSpin** must be chosen appropriately. For non-Γ point calculations, the number of electrons is set to be equal at all k points. **NonCollinearSpin** calculations are not supported at present.

It is important to note that the OMM requires all occupied Kohn-Sham eigenvalues to be negative; this can be achieved by applying a shift to the eigenspectrum, controlled by **ON.eta** (in this case, **ON.eta** simply needs to be higher than the HOMO level). If the OMM exhibits a pathologically slow or unstable convergence, this is almost certainly due to the fact that the default value of **ON.eta** (`0.0 eV`) is too low, and should be raised by a few eV.

OMM.UseCholesky (*logical*): Select whether to perform a Cholesky factorization of the generalized eigenvalue problem; this removes the overlap matrix from the problem but also destroys the sparsity of the Hamiltonian matrix.

Use: Used only if **SolutionMethod** = `OMM`

Default value: `.true.`

OMM.Use2D (*logical*): Select whether to use a 2D data decomposition of the matrices for parallel calculations. This generally leads to superior scaling for large numbers of MPI processes.

Use: Used only if **SolutionMethod** = `OMM`

Default value: `.true.`

OMM.UseSparse (*logical*): Select whether to make use of the sparsity of the Hamiltonian and overlap matrices where possible when performing matrix-matrix multiplications (these operations are thus reduced from $O(N^3)$ to $O(N^2)$ without loss of accuracy).

Use: Used only if **SolutionMethod** = `OMM`; not compatible with **OMM.UseCholesky**, **OMM.Use2D**, or non-Γ point calculations

Default value: `.false.`

OMM.Precon (*integer*): Number of SCF steps for *all* MD steps for which to apply a preconditioning scheme based on the overlap and kinetic energy matrices; for negative values the preconditioning is always applied. Preconditioning is usually essential for fast and accurate convergence (note, however, that it is not needed if a Cholesky factorization is performed; in such cases this variable will have no effect on the calculation).

Use: Used only if **SolutionMethod** = `OMM`; not used with **OMM.UseCholesky**

Default value: `-1`

OMM.PreconFirstStep (*integer*): Number of SCF steps in the *first* MD step for which to apply the preconditioning scheme; if present, this will overwrite the value given in **OMM.Precon** for the first MD step only.

Use: Used only if **SolutionMethod** = `OMM`; not used with **OMM.UseCholesky**

Default value: `OMM.Precon`

OMM.Diagon (*integer*): Number of SCF steps for *all* MD steps for which to use a standard diagonalization before switching to the OMM; for negative values diagonalization is always used, and so the calculation is effectively equivalent to **SolutionMethod** = `diagon`. In general, selecting the first few SCF steps can speed up the calculation by removing the costly initial minimization (at present this works best for Γ point calculations).

Use: Used only if **SolutionMethod** = `OMM`

Default value: `0`

OMM.DiagonFirstStep (*integer*): Number of SCF steps in the *first* MD step for which to use a standard diagonalization before switching to the OMM; if present, this will overwrite the value given in **OMM.Diagon** for the first MD step only.

Use: Used only if **SolutionMethod** = `OMM`

Default value: `OMM.Diagon`

OMM.BlockSize (*integer*): Blocksize used for distributing the elements of the matrix over MPI processes. Specifically, this variable controls the dimension relating to the trial orbitals used in the minimization (equal to the number of occupied states at each k point/spin); the equivalent variable for the dimension relating to the underlying basis orbitals is controlled by **BlockSize**.

Use: Used only if **SolutionMethod** = `OMM`

Default value: chosen by SIESTA

OMM.TPreconScale (*real energy*): Scale of the kinetic energy preconditioning (see C. K. Gan *et al.*, Comput. Phys. Commun. **134**, 33 (2001)). A smaller value indicates more aggressive kinetic energy preconditioning, while an infinite value indicates no kinetic energy preconditioning. In general, the kinetic energy preconditioning is much less important than the tensorial correction brought about by the overlap matrix, and so this value will have fairly little impact on the overall performace of the preconditioner; however, too aggressive kinetic energy preconditioning can have a detrimental effect on performance and accuracy.

Use: Used only if **SolutionMethod** = `OMM`

Default value: `10.0 Ry`

OMM.RelTol (*real*): Relative tolerance in the conjugate gradients minimization of the Kohn-Sham band energy (see **ON.etol**).

Use: Used only if **SolutionMethod** = `OMM`

Default value: 10^{-9}

OMM.Eigenvalues (*logical*): Select whether to perform a diagonalization at the end of each MD step to obtain the Kohn-Sham eigenvalues.

Use: Used only if **SolutionMethod** = `OMM`

Default value: `.false.`

OMM.WriteCoeffs (*logical*): Select whether to write the coefficients of the solution orbitals to file at the end of each MD step.

Use: Used only if **SolutionMethod** = `OMM`

Default value: `.false.`

OMM.ReadCoeffs (*logical*): Select whether to read the coefficients of the solution orbitals from file at the beginning of a new calculation. Useful for restarting an interrupted calculation, especially when used in conjuction with **DM.UseSaveDM**. Note that the same number of MPI processes and values of **OMM.Use2D**, **OMM.BlockSize**, and **BlockSize** must be used when restarting.

Use: Used only if **SolutionMethod** = `OMM`

Default value: `.false.`

OMM.LongOutput (*logical*): Select whether to output detailed information of the conjugate gradients minimization for each SCF step.

Use: Used only if **SolutionMethod** = `OMM`

Default value: `.false.`

6.11.5 Order(N) calculations

The Ordern(N) subsystem is quite fragile and only works for systems with clearly separated occupied and empty states. Note also that the option to compute the chemical potential automatically does not yet work in parallel.

ON.functional (*string*): Choice of order-N minimization functionals:

- `Kim`: Functional of Kim, Mauri and Galli, PRB 52, 1640 (1995).
- `Ordejon-Mauri`: Functional of Ordejón et al, or Mauri et al, see PRB 51, 1456 (1995). The number of localized wave functions (LWFs) used must coincide with $N_{el}/2$ (unless spin polarized). For the initial assignment of LWF centers to atoms, atoms with even number of electrons, n, get n/2 LWFs. Odd atoms get (n+1)/2 and (n-1)/2 in an alternating sequence, ir order of appearance (controlled by the input in the atomic coordinates block).
- `files`: Reads localized-function information from a file and chooses automatically the functional to be used.

Use: Used only if **SolutionMethod** = `ordern`

Default value: `Kim`

ON.MaxNumIter (*integer*): Maximum number of iterations in the conjugate minimization of the electronic energy, in each SCF cycle.

Use: Used only if **SolutionMethod** = `OrderN`

Default value: `1000`

ON.etol (*real*): Relative-energy tolerance in the conjugate minimization of the electronic energy. The minimization finishes if $2(E_n - E_{n-1})/(E_n + E_{n-1}) \leq$ ON.etol.

Use: Used only if **SolutionMethod** = `OrderN`

Default value: 10^{-8}

ON.eta (*real energy*): Fermi level parameter of Kim *et al.*. This should be in the energy gap, and tuned to obtain the correct number of electrons. If the calculation is spin polarised, then separate Fermi levels for each spin can be specified.

Use: Used only if **SolutionMethod** = `OrderN`

Default value: `0.0 eV`

ON.eta_alpha (*real energy*): Fermi level parameter of Kim *et al.* for alpha spin electrons. This should be in the energy gap, and tuned to obtain the correct number of electrons. Note that if the Fermi level is not specified individually for each spin then the same global eta will be used.

Use: Used only if **SolutionMethod** = `OrderN`

Default value: `0.0 eV`

ON.eta_beta (*real energy*): Fermi level parameter of Kim *et al.* for beta spin electrons. This should be in the energy gap, and tuned to obtain the correct number of electrons. Note that if the Fermi level is not specified individually for each spin then the same global eta will be used.

Use: Used only if **SolutionMethod** = `OrderN`

Default value: `0.0 eV`

ON.RcLWF (*real legth*): Localization redius for the Localized Wave Functions (LWF's).

Use: Used only if **SolutionMethod** = `OrderN`

Default value: `9.5 Bohr`

ON.ChemicalPotential (*logical*): Specifies whether to calculate an order-N estimate of the Chemical Potential, by the projection method (Goedecker and Teter, PRB **51**, 9455 (1995); Stephan, Drabold and Martin, PRB **58**, 13472 (1998)). This is done by expanding the Fermi function (or density matrix) at a given temperature, by means of Chebyshev polynomials, and imposing a real space truncation on the density matrix. To obtain a realistic estimate, the temperature should be small enough (typically, smaller than the energy gap), the localization range large enough (of the order of the one you would use for the Localized Wannier Functions), and the order of the polynomial expansion sufficiently large (how large depends on the temperature; typically, 50-100).

Use: Used only if **SolutionMethod** = `OrderN`.

Default value: `.false.`

Note: This option does not work in parallel. An alternative is to obtain the approximate value of the chemical potential using an initial diagonalization.

ON.ChemicalPotentialUse (*logical*): Specifies whether to use the calculated estimate of the Chemical Potential, instead of the parameter **ON.eta** for the order-N energy functional minimization. This is useful if you do not know the position of the Fermi level, typically in the beginning of an order-N run.

Use: Used only if **SolutionMethod** = `OrderN`. Overrides the value of **ON.eta**. Overrides the value of **ON.ChemicalPotential**, setting it to `.true.`.

Default value: `.false.`

Note: This option does not work in parallel. An alternative is to obtain the approximate value of the chemical potential using an initial diagonalization.

ON.ChemicalPotentialRc (*real length*): Defines the cutoff radius for the density matrix or Fermi operator in the calculation of the estimate of the Chemical Potential.

Use: Used only if **SolutionMethod** = `OrderN` and **ON.ChemicalPotential** or **ON.ChemicalPotentialUse** = `.true.`

Default value: `9.5 Bohr`.

ON.ChemicalPotentialTemperature (*real temperature or energy*): Defines the temperature to be used in the Fermi function expansion in the calculation of the estimate of the Chemical Potential. To have an accurate results, this temperature should be smaller than the gap of the system.

Use: Used only if **SolutionMethod** = `OrderN`, and **ON.ChemicalPotential** or **ON.ChemicalPotentialUse** = `.true.`

Default value: `0.05 Ry`.

ON.ChemicalPotentialOrder (*integer*): Order of the Chebishev expansion to calculate the estimate of the Chemical Potential.

Use: Used only if **SolutionMethod** = `OrderN`, and **ON.ChemicalPotential** or **ON.ChemicalPotentialUse** = `.true.`

Default value: `100`

ON.LowerMemory (*logical*): If .true., then a slightly reduced memory algorithm is used in the 3-point line search during the order N minimisation. Only affects parallel runs.

Use: Used only if **SolutionMethod** = `OrderN`

Default value: `.false.`

Output of localized wavefunctions

At the end of each conjugate gradient minimization of the energy functional, the LWF's are stored on disk. These can be used as an input for the same system in a restart, or in case something goes wrong. The LWF's are stored in sparse form in file SystemLabel.LWF

It is important to keep very good care of this file, since the first minimizations can take MANY steps. Loosing them will mean performing the whole minimization again. It is also a good practice to save it periodically during the simulation, in case a mid-run restart is necessary.

ON.UseSaveLWF (*logical*): Instructs to read the localized wave functions stored in file `SystemLabel`.LWF by a previous run.

Use: Used only if **SolutionMethod** is `OrderN`. If the required file does not exist, a warning is printed but the program does not stop. Overrides **UseSaveData**.

Default value: `.false.`

6.12 Band-structure analysis

This calculation of the band structure is performed optionally after the geometry loop finishes, and the output information written to the SystemLabel.bands file (see below for the format).

BandLinesScale (*string*): Specifies the scale of the k vectors given in **BandLines** and **BandPoints** below. The options are:

- `pi/a` (k-vector coordinates are given in Cartesian coordinates, in units of π/a, where a is the lattice constant)
- `ReciprocalLatticeVectors` (k vectors are given in reciprocal-lattice-vector coordinates)

Default value: `pi/a`

Note: You might need to define explicitly a LatticeConstant tag in your fdf file if you do not already have one, and make it consistent with the scale of the k-points and any unit-cell vectors you might have already defined.

BandLines (*data block*): Specifies the lines along which band energies are calculated (usually along high-symmetry directions). An example for an FCC lattice is:

```
%block BandLines
  1  1.000  1.000  1.000  L        # Begin at L
 20  0.000  0.000  0.000  \Gamma    # 20 points from L to gamma
 25  2.000  0.000  0.000  X        # 25 points from gamma to X
 30  2.000  2.000  2.000  \Gamma    # 30 points from X to gamma
%endblock BandLines
```

where the last column is an optional LaTeX label for use in the band plot. If only given points (not lines) are required, simply specify 1 in the first column of each line. The first column of the first line must be always 1.

Use: Used only if **SolutionMethod** = `diagon`. The band k points are unrelated and compatible with any k-grid used to calculate the total energy and charge density. This block is overriden by **BandPoints** if both are present.

Default value: No band energies calculated.

BandPoints (*data block*): Band energies are calculated for the list of arbitrary k points given in the block. Units defined by **BandLinesScale** as for **BandLines**. The generated *Systemlabel*`.bands` file will contain the k point coordinates (in a.u.) and the corresponding band energies (in eV). Example:

```
%block BandPoints
   0.000  0.000  0.000   # This is a comment. eg this is gamma
   1.000  0.000  0.000
   0.500  0.500  0.500
%endblock BandPoints
```

Use: Used only if **SolutionMethod** = `diagon`. The band k points are unrelated and compatible with any k-grid used to calculate the total energy and charge density. If both are present, this block supersedes **BandLines**.

Default value: No band energies calculated.

WriteKbands (*logical*): If `.true.` it writes the coordinates of the $\vec{k}$ vectors defined for band plotting, to the main output file.

Use: Only if **SolutionMethod** is `diagon`.

Default value: `.false.` (see **LongOutput**)

WriteBands (*logical*): If `.true.` it writes the Hamiltonian eigenvalues corresponding to the $\vec{k}$ vectors defined for band plotting, in the main output file.

Use: Only if **SolutionMethod** is `diagon`.

Default value: `.false.` (see **LongOutput**)

6.12.1 Format of the .bands file

```
FermiEnergy (all energies in eV) \\
kmin, kmax (along the k-lines path, i.e. range of k in the band plot) \\
Emin, Emax (range of all eigenvalues) \\
NumberOfBands, NumberOfSpins (1 or 2), NumberOfkPoints \\
k1, ((ek(iband,ispin,1),iband=1,NumberOfBands),ispin=1,NumberOfSpins) \\
k2, ek \\
 . \\
 . \\
 . \\
klast, ek \\
NumberOfkLines \\
kAtBegOfLine1, kPointLabel \\
kAtEndOfLine1, kPointLabel \\
  . \\
  . \\
  . \\
kAtEndOfLastLine, kPointLabel \\
```

The GNUBANDS postprocessing utility program (found in the Util/Bands directory) reads the *Systemlabel*.bands for plotting. See the **BandLines** data descriptor above for more information.

6.12.2 Output of wavefunctions associated to bands

The user can optionally request that the wavefunctions corresponding to the computed bands be written to file. They are written to the *SystemLabel*.bands.WFSX file . The relevant options are:

WFS.Write.For.Bands (*logical*): Instructs the program to compute and write the wave functions associated to the bands specified (by a **BandLines** or a **BandPoints** block) to the file `SystemLabel.bands`.WFSX.

The information in this file might be useful, among other things, to generate "fatbands" plots, in which both band eigenvalues and information about orbital projections is presented. See the `fat` program in the `Util/COOP` directory for details.

Default value: `.false.`

WFS.Band.Min (*integer*): Specifies the lowest band index of the wave-functions to be written to the file `SystemLabel.bands`.WFSX for each k-point (all k-points in the band set are affected).

Default value: `1`

WFS.Band.Max (*integer*): Specifies the highest band index of the wave-functions to be written to the file `SystemLabel.bands`.WFSX for each k-point (all k-points in the band set are affected).

Default value: `(the number of orbitals)`

6.13 Output of selected wavefunctions

The user can optionally request that specific wavefunctions are written to file. These wavefunctions are re-computed after the geometry loop (if any) finishes, using the last (presumably converged) density matrix produced during the last self-consistent field loop (after a final mixing). They are written to the SystemLabel.selected.WFSX file.

Note that the complete set of wavefunctions obtained during the last iteration of the SCF loop will be written to SystemLabel.fullBZ.WFSX if the **COOP.write** option is in effect.

Note that the complete set of wavefunctions obtained during the last iteration of the SCF loop will be written to a NetCDF file `WFS.nc` if the **Diag.UseNewDiagk** option is in effect.

WaveFuncKPointsScale (*string*): Specifies the scale of the k vectors given in **WaveFuncKPoints** below. The options are:

- `pi/a` (k-vector coordinates are given in Cartesian coordinates, in units of π/a, where a is the lattice constant)
- `ReciprocalLatticeVectors` (k vectors are given in reciprocal-lattice-vector coordinates)

Default value: `pi/a`

WaveFuncKPoints (*data block*): Specifies the k-points at which the electronic wavefunction coefficients are written. An example for an FCC lattice is:

```
%block WaveFuncKPoints
0.000  0.000  0.000  from 1 to 10   # Gamma wavefuncs 1 to 10
2.000  0.000  0.000  1 3 5          # X wavefuncs 1,3 and 5
```

```
1.500  1.500  1.500                  # K wavefuncs, all
%endblock WaveFuncKPoints
```

The index of a wavefunction is defined by its energy, so that the first one has lowest energy.

The user can also narrow the energy-range used with the **WFS.Energy.Min** and **WFS.Energy.Max** options (both take an energy (with units) as extra argument – see section 6.15.3). Care should be taken to make sure that the actual values of the options make sense.

Use: Used only if **SolutionMethod** = `diagon`. These k points are unrelated and compatible with any k-grid used to calculate the total energy, charge density and band structure.

Default value: No wavefunctions are written.

WriteWaveFunctions (*logical*): If `.true.` it writes to the output file a list of the wavefunctions actually written to the *Systemlabel*.selected.WFSX file, which is always produced.

Use: Only if **SolutionMethod** is `diagon`.

Default value: `.false.` (see **LongOutput**)

The unformatted WFSX file contains the information of the k-points for which wavefunctions coefficients are written, and the energies and coefficients of each wavefunction which was specified in the input file (see **WaveFuncKPoints** descriptor above). It also contains information on the atomic species and the orbitals for postprocessing purposes.

NOTE: The WFSX file is in a more compact form than the old WFS, and the wavefunctions are output in single precision. The `Util/WFS/wfsx2wfs` program can be used to convert to the old format.

The readwf and readwfsx postprocessing utilities programs (found in the Util/WFS directory) read the WFS or WFSX files, respectively, and generate a readable file.

6.14 Densities of states

6.14.1 Total density of states

There are several options to obtain the total density of states:

- The Hamiltonian eigenvalues for the SCF sampling $\vec{k}$ points can be dumped into SystemLabel.EIG in a format analogous to SystemLabel.bands, but without the kmin, kmax, emin, emax information, and without the abscissa. The Eig2DOS postprocessing utility can be then used to obtain the density of states. See the **WriteEigenvalues** descriptor.
- As a side-product of a partial-density-of-states calculation (see below)
- As one of the files produced by the `Util/COOP/mprop` during the off-line analysis of the electronic structure. This method allows the flexibility of specifying energy ranges and resolutions at will, without re-running Siesta See Sec. 6.15.3.

6.14.2 Partial (projected) density of states

There are two options to obtain the partial density of states

- Using the options below
- Using the `Util/COOP/mprop` program for the off-line analysis of the electronic structure in PDOS mode. This method allows the flexibility of specifying energy ranges, orbitals, and resolutions at will, without re-running SIESTA. See Sec. 6.15.3.

ProjectedDensityOfStates (*block*):

Instructs to write the Total Density Of States (Total DOS) and the Projected Density Of States (PDOS) on the basis orbitals, between two given energies, in files `SystemLabel`.DOS and `SystemLabel`.PDOS, respectively. The block must be a single line with the energies of the range for PDOS projection, (relative to the program's zero, i.e. the same as the eigenvalues printed by the program), the peak width (an energy) for broadening the eigenvalues, the number of points in the energy window, and the energy units. An example is:

```
%block ProjectedDensityOfStates
   -20.00  10.00  0.200  500  eV
%endblock ProjectedDensityOfStates
```

By default the projected density of states is generated for the same grid of points in reciprocal space as used for the SCF calculation. However, a separate set of K-points, usually on a finer grid, can be generated using one of the options **PDOS.kgrid_cutoff** or **PDOS.kgrid_Monkhorst_Pack**. The format of these options is exactly the same as for **kgrid_cutoff** and **kgrid_Monkhorst_Pack**, respectively. Note that if a gamma point calculation is being used in the SCF part, especially as part of a geometry optimisation, and this is then to be run with a grid of K-points for the PDOS calculation it is more efficient to run the SCF phase first and then restart to perform the PDOS evaluation using the density matrix saved from the SCF phase.

Use: The two energies of the range must be ordered, with lowest first.

Output: The Total DOS is dumped into a file called `SystemLabel`.DOS. The format of this file is:

Energy value, Total DOS (spin up), Total DOS (spin down)

The Projected Density Of States for all the orbitals in the unit cell is dumped sequentially into a file called `SystemLabel`.PDOS. This file is structured using spacing and xml tags. A machine-readable (but not very human readable) xml file `SystemLabel.PDOS.xml` is also produced. Both can be processed by the program in `Util/pdosxml`. The .PDOS file can be processed by utilites in `Util/Contrib/APostnikov`.

In all cases, the units for the DOS are (number of states/eV), and the Total DOS, $g(\epsilon)$, is normalized as follows:

$$\int_{-\infty}^{+\infty} g(\epsilon)\, d\epsilon = number\ \ of\ \ basis\ \ orbitals\ \ in\ \ unit\ \ cell \tag{4}$$

Default value: PDOS not calculated nor written.

6.14.3 Local density of states

The LDOS is formally the DOS weighted by the amplitude of the corresponding wavefunctions at different points in space, and is then a function of energy and position. SIESTA can output the LDOS integrated over a range of energies. This information can be used to obtain simple STM images in the Tersoff-Hamann approximation (See `Util/STM/simple-stm`).

LocalDensityOfStates (*block*): Instructs to write the LDOS, integrated between two given energies, at the mesh used by DHSCF, in file `SystemLabel`.LDOS. This file can be read by routine IORHO, which may be used by an application program in later versions. The block must be a single line with the energies of the range for LDOS integration (relative to the program's zero, i.e. the same as the eigenvalues printed by the program) and their units. An example is:

```
%block LocalDensityOfStates
   -3.50  0.00   eV
%endblock LocalDensityOfStates
```

Use: The two energies of the range must be ordered, with lowest first. File `SystemLabel`.LDOS is only written, not read, by siesta.

Default value: LDOS not calculated nor written.

6.15 Options for chemical analysis

6.15.1 Mulliken charges and overlap populations

WriteMullikenPop (*integer*): It determines the level of Mulliken population analysis printed:

- 0 = None
- 1 = atomic and orbital charges
- 2 = 1 + atomic overlap pop.
- 3 = 2 + orbital overlap pop.

The order of the orbitals in the population lists is defined by the order of atoms. For each atom, populations for PAO orbitals and double-z, triple-z, etc... derived from them are displayed first for all the angular momenta. Then, populations for perturbative polarization orbitals are written. Within a l-shell be aware that the order is not conventional, being y, z, x for p orbitals, and xy, yz, z^2, xz, and $x^2 - y^2$ for d orbitals.

Default value: `0` (see **LongOutput**)

MullikenInSCF (*logical*): If true, the Mulliken populations will be written for every SCF step at the level of detail specified in **WriteMullikenPop**. Useful when dealing with SCF problems, otherwise too verbose.

Default value: `.false.`

SpinInSCF (*logical*):

If true, the size and components of the (total) spin polarization will be printed at every scf step. This is analogous to the 'Mulliken.In.Scf' feature.

Default value: `.true.` for calculations involving spin.

6.15.2 Voronoi and Hirshfeld atomic population analysis

WriteHirshfeldPop (*logical*): If `.true.`, the program calculates and prints the Hirshfeld **net** atomic populations on each atom in the system. For a definition of the Hirshfeld charges, see Hirshfeld, Theo Chem Acta **44**, 129 (1977) and Fonseca et al, J. Comp. Chem. **25**, 189 (2003). Hirshfeld charges are more reliable than Mulliken charges, specially for large basis sets. The number printed is the total net charge of the atom: the variation from the neutral charge, in units of $|e|$: positive (negative) values indicate deficiency (excess) of electrons in the atom.

Default value: `.false.`

WriteVoronoiPop (*logical*): If `.true.`, the program calculates and prints the Voronoi **net** atomic populations on each atom in the system. For a definition of the Voronoi charges, see Bickelhaupt et al, Organometallics **15**, 2923 (1996) and Fonseca et al, J. Comp. Chem. **25**, 189 (2003). Voronoi charges are more reliable than Mulliken charges, specially for large basis sets. The number printed is the total net charge of the atom: the variation from the neutral charge, in units of $|e|$: positive (negative) values indicate deficiency (excess) of electrons in the atom.

Default value: `.false.`

The Hirshfeld and Voronoi populations (partial charges) are computed by default only at the end of the program (i.e., for the final geometry, after self-consistency). The following options allow more control:

PartialChargesAtEveryGeometry (*logical*):

The Hirshfeld and Voronoi populations are computed after self-consistency is achieved, for all the geometry steps.

Default value: `.false.`

PartialChargesAtEveryScfStep (*logical*):

The Hirshfeld and Voronoi populations are computed for every step of the self-consistency process.

Default value: `.false.`

Performance note: The default behavior (computing at the end of the program) involves an extra calculation of the charge density.

6.15.3 Crystal-Orbital overlap and hamilton populations (COOP/COHP)

These curves are quite useful to analyze the electronic structure to get insight about bonding characteristics. See the `Util/COOP` directory for more details. The **COOP.Write** option must be activated to get the information needed.

References:

- Original COOP reference: Hughbanks, T.; Hoffmann, R., J. Am. Chem. Soc., 1983, 105, 3528.
- Original COHP reference: Dronskowski, R.; Blchl, P. E., J. Phys. Chem., 1993, 97, 8617.
- A tutorial introduction: Dronskowski, R. Computational Chemistry of Solid State Materials; Wiley-VCH: Weinheim, 2005.
- Online material maintained by R. Dronskowski's group: `http://www.cohp.de/`

COOP.Write (*logical*): Instructs the program to generate `SystemLabel.fullBZ`.WFSX (packed wavefunction file) and `SystemLabel`.HSX (H, S and X_ ij file), to be processed by `Util/COOP/mprop` to generate COOP/COHP curves, (projected) densities of states, etc.

The WFSX file is in a more compact form than the usual WFS, and the wavefunctions are output in single precision. The `Util/wfsx2wfs` program can be used to convert to the old format. The HSX file is in a more compact form than the usual HS, and the Hamiltonian, overlap matrix, and relative-positions array (which is always output, even for gamma-point only calculations) are in single precision.

Default value: `.false.`

The user can narrow the energy-range used (and save some file space) by using the **WFS.Energy.Min** and **WFS.Energy.Max** options (both take an energy (with units) as extra argument), and/or the **WFS.Band.Min** and **WFS.Band.Max** options. Care should be taken to make sure that the actual values of the options make sense.

Note that the band range options could also affect the output of wave-functions associated to bands (see section 6.12.2), and that the energy range options could also affect the output of user-selected wave-functions with the **WaveFuncKpoints** block (see section 6.13).

WFS.Energy.Min (*physical energy*): Specifies the lowest value of the energy (eigenvalue) of the wave-functions to be written to the file `SystemLabel.fullBZ`.WFSX for each k-point (all k-points in the BZ sampling are affected).

Default value: $-\infty$

WFS.Energy.Max (*physical energy*): Specifies the highest value of the energy (eigenvalue) of the wave-functions to be written to the file `SystemLabel.fullBZ`.WFSX for each k-point (all k-points in the BZ sampling are affected).

Default value: $+\infty$

WFS.Band.Min (*integer*): Specifies the lowest band index of the wave-functions to be written to the file (in this context) `SystemLabel.fullBZ`.WFSX for each k-point (all k-points in the BZ sampling are affected).

Default value: `1`

WFS.Band.Max (*integer*): Specifies the highest band index of the wave-functions to be written to the file (in this context) `SystemLabel.fullBZ`.WFSX for each k-point (all k-points in the BZ sampling are affected).

Default value: `(the number of orbitals)`

6.16 Optical properties

OpticalCalculation (*logical*): If specified, the imaginary part of the dielectric function will be calculated and stored in a file called *Systemlabel*.**EPSIMG**. The calculation is performed using the simplest approach based on the dipolar transition matrix elements between different eigenfunctions of the self-consistent Hamiltonian. For molecules the calculation is performed using the position operator matrix elements, while for solids the calculation is carried out in the momentum space formulation. Corrections due to the non-locality of the pseudopotentials are introduced in the usual way.

Default value: `false`

Optical.EnergyMinimum (*real energy*): This specifies the minimum of the energy range in which the frequency spectrum will be calculated.

Default value: 0 Ry.

Optical.EnergyMaximum (*real energy*): This specifies the maximum of the energy range in which the frequency spectrum will be calculated.

Default value: 10 Ry.

Optical.Broaden (*real energy*): If this is value is set then a Gaussian broadening will be applied to the frequency values.

Default value: 0 Ry.

Optical.Scissor (*real energy*): Because of the tendency of DFT calculations to under estimate the band gap, a rigid shift of the unoccupied states, known as the scissor operator, can be added to correct the gap and thereby improve the calculated results. This shift is only applied to the optical calculation and no where else within the calculation.

Default value: 0 Ry.

Optical.NumberOfBands (*integer*): This option controls the number of bands that are included in the optical property calculation. Clearly this number must be larger than the number of occupied bands and less than or equal to the number of basis functions (which determines the number of unoccupied bands available). Note, while including all the bands may be the most accurate choice this will also be the most expensive!

Default value: All bands.

Optical.Mesh (*data block*): This block contains 3 numbers that determine the mesh size used for the integration across the Brillouin zone. For example:

```
%block  Optical.Mesh
  5 5 5
%endblock  Optical.Mesh
```

The three values represent the number of mesh points in the direction of each reciprocal lattice vector.

Default value: Empty in general. For atoms or molecules a k-sampling of only one point is assumed.

Optical.OffsetMesh (*logical*): If set to true, then the mesh is offset away from the gamma point for odd numbers of points.

Default value: `false`

Optical.PolarizationType (*string*): This option has three possible values that represent the type of polarization to be used in the calculation. The options are **polarized**, which implies the application of an electric field in a given direction, **unpolarized**, which implies the propagation of light in a given direction, and **polycrystal**. In the case of the first two options a direction in space must be specified for the electric field or propagation using the *Optical.Vector* data block.

Default value: `polycrystal`

Optical.Vector (*data block*): This block contains 3 numbers that specify the vector direction for either the electric field or light propagation, for a polarized or unpolarized calculation, respectively. A typical block might look like:

```
%block  Optical.Vector
  1.0 0.0 0.5
%endblock  Optical.Vector
```

Default value: Empty.

6.17 Macroscopic polarization

PolarizationGrids (*data block*): If specified, the macroscopic polarization will be calculated using the geometric Berry phase approach (R.D. King-Smith, and D. Vanderbilt, PRB **47**, 1651 (1993)). In this method the electronic contribution to the macroscopic polarization, along a given direction, is calculated using a discretized version of the formula

$$P_{e,\parallel} = \frac{i f q_e}{8\pi^3} \int_A d\mathbf{k}_\perp \sum_{n=1}^{M} \int_0^{|G_\parallel|} dk_\parallel \langle u_{\mathbf{k}n} | \frac{\delta}{\delta k_\parallel} | u_{\mathbf{k}n} \rangle \tag{5}$$

where f is the occupation (2 for a non-magnetic system), q_e the electron charge, M is the number of occupied bands (the system **must** be an insulator), and $u_{\mathbf{k}n}$ are the periodic Bloch functions. $\mathbf{G}_\parallel$ is the shortest reciprocal vector along the chosen direction.

As it can be seen in formula (5), to compute each component of the polarization we must perform a surface integration of the result of a 1-D integral in the selected direction. The grids for the calculation along the direction of each of the three lattice vectors are specified in the block **PolarizationGrids**.

```
%block PolarizationGrids
   10   3  4      yes
    2  20  2       no
    4   4 15
%endblock PolarizationGrids
```

All three grids must be specified, therefore a 3×3 matrix of integer numbers must be given: the first row specifies the grid that will be used to calculate the polarization along the direction of the first lattice vector, the second row will be used for the calculation along the the direction of the second lattice vector, and the third row for the third lattice vector. The numbers in the diagonal of the matrix specifie the number of points to be used in the one dimensional line integrals along the different directions. The other numbers specifie the mesh used in the surface integrals. The last column specifies if the bidimensional grids are going to be diplaced from the origin or not, as in the Monkhorst-Pack algorithm (PRB **13**, 5188 (1976)). This last column is optional. If the number of points in one of the grids is zero, the calculation will not be performed for this particular direction.

For example, in the given example, for the computation in the direction of the first lattice vector, 15 points will be used for the line integrals, while a 3×4 mesh will be used for the surface integration. This last grid will be displaced from the origin, so Γ will not be included in the bidimensional integral. For the directions of the second and third lattice vectors, the number of points will be 20 and 2×2, and 15 and 4×4, respectively.

It has to be stressed that the macroscopic polarization can only be meaningfully calculated using this approach for insulators. Therefore, the presence of an energy gap is necessary, and no band can cross the Fermi level. The program performs a simple check of this condition, just by counting the electrons in the unit cell (the number must be even for a non-magnetic system, and the total spin polarization must have an integer value for spin polarized systems), however is the responsability of the user to check that the system under study is actually an insulator (for both spin components if spin polarized).

The total macroscopic polarization, given in the output of the program, is the sum of the electronic contribution (calculated as the Berry phase of the valence bands), and the ionic contribution, which is simply defined as the sum of the atomic positions within the unit cell multiply by the ionic charges ($\sum_i^{N_a} Z_i \mathbf{r}_i$). In the case of the magnetic systems, the bulk polarization for each spin component has been defined as

$$\mathbf{P}^\sigma = \mathbf{P}_e^\sigma + \frac{1}{2}\sum_i^{N_a} Z_i \mathbf{r}_i \tag{6}$$

N_a is the number of atoms in the unit cell, and $\mathbf{r}_i$ and Z_i are the positions and charges of the ions.

It is also worth noting, that the macroscopic polarization given by formula (5) is only defined modulo a "quantum" of polarization (the bulk polarization per unit cell is only well defined modulo $fq_e\mathbf{R}$, being $\mathbf{R}$ an arbitrary lattice vector). However, the experimentally observable quantities are associated to changes in the polarization induced by changes on the atomic positions (dynamical charges), strains (piezoelectric tensor), etc... The calculation of those changes, between different configurations of the solid, will be well defined as long as they are smaller than the "quantum", i.e. the perturbations are small enough to create small changes in the polarization.

Use: Only compatible with **SolutionMethod** = diagon.
Default value: Empty. No calculation performed.

BornCharge (*logical*): If true, the Born effective charge tensor is calculated for each atom by finite differences, by calculating the change in electric polarization (see **PolarizationGrids**) induced by the small displacements generated for the force constants calculation (see **MD.TypeOfRun** = `FC`):

$$Z^*_{i,\alpha,\beta} = \frac{\Omega_0}{e}\left.\frac{\partial P_\alpha}{\partial u_{i,\beta}}\right|_{q=0} \tag{7}$$

where e is the charge of an electron and Ω_0 is the unit cell volume.

To calculate the Born charges it is necessary to specify both the Born charge flag and the mesh used to calculate the polarization, for example:

```
%block PolarizationGrids
7  3  3
3  7  3
3  3  7
%endblock PolarizationGrids
BornCharge True
```

The Born effective charge matrix is then written to the file *SystemLabel*`.BC`.

The method by which the polarization is calculated may introduce an arbitrary phase (polarization quantum), which in general is far larger than the change in polarization which results from the atomic displacement. It is removed during the calculation of the Born effective charge tensor.

The Born effective charges allow the calculation of LO-TO splittings and infrared activities. The version of the Vibra utility code in which these magnitudes are calculated is not yet distributed with SIESTA, but can be obtained form Tom Archer (archert@tcd.ie).

Use: Only used if **MD.TypeOfRun** is `FC`.

Default value: `false`

6.18 Maximally Localized Wannier Functions. Interface with the WANNIER90 code

WANNIER90 (http://www.wannier.org) is a code to generate maximally localized wannier functions according to the original Marzari and Vanderbilt recipe.

It is strongly recommended to read the original papers on which this method is based and the documentation of the WANNIER90 code. Here we shall focus only on those internal SIESTA variables required to produce the files that will be processed by WANNIER90.

A complete list of examples and tests (including molecules, metals, semiconductors, insulators, magnetic systems, plotting of Fermi surfaces or interpolation of bands), can be downloaded from

http://personales.unican.es/junqueraj/Wannier-examples.tar.gz

NOTE: The Bloch functions produced by a first-principles code have arbitrary phases that depend on the number of processors used and other possibly non-reproducible details of the calculation. In what follows it is essential to maintain consistency in the handling of the overlap and Bloch-funcion files produced and fed to WANNIER90.

Siesta2Wannier90.WriteMmn (*logical*):

This flag determines whether the overlaps between the periodic part of the Bloch states at neighbour k-points are computed and dumped into a file in the format required by WANNIER90. These overlaps are defined in Eq. (27) in the paper by N. Marzari *et al.*, Review of Modern Physics **84**, 1419 (2012), or Eq. (1.7) of the Wannier90 User Guide, Version 2.0.1.

The k-points for which the overlaps will be computed are read from a `.nnkp` file produced by WANNIER90. It is strongly recommended for the user to read the corresponding user guide.

The overlap matrices are written in a file with extension `.mmn`.

Default value: `.false.`

Siesta2Wannier90.WriteAmn (*logical*):

This flag determines whether the overlaps between Bloch states and trial localized orbitals are computed and dumped into a file in the format required by WANNIER90. These projections are defined in Eq. (16) in the paper by N. Marzari *et al.*, Review of Modern Physics **84**, 1419 (2012), or Eq. (1.8) of the Wannier90 User Guide, Version 2.0.1.

The localized trial functions to use are taken from the `.nnkp` file produced by WANNIER90. It is strongly recommended for the user to read the corresponding user guide.

The overlap matrices are written in a file with extension `.amn`.

Default value: `.false.`

Siesta2Wannier90.WriteEig (*logical*):

Flag that determines whether the Kohn-Sham eigenvalues (in eV) at each point in the Monkhorst-Pack mesh required by WANNIER90 are written to file. This file is mandatory in WANNIER90 if any of disentanglement, plot_bands, plot_fermi_surface or hr_plot options are set to true in the WANNIER90 input file.

The eigenvalues are written in a file with extension `.eigW`. This extension is chosen to avoid name clashes with SIESTA's standard eigenvalue file in case-insensitive filesystems.

Default value: `.false.`

Siesta2Wannier90.WriteUnk (*logical*):

Produces `UNKXXXXX.Y` files which contain the periodic part of a Bloch function in the unit cell on a grid given by global unk_nx, unk_ny, unk_nz variables. The name of the output files is assumed to have the previous form, where the `XXXXXX` refer to the k-point index (from 00001 to the total number of k-points considered), and the `Y` refers to the spin component (1 or 2)

The periodic part of the Bloch functions is defined by

$$u_{n\vec{k}}(\vec{r}) = \sum_{\vec{R}\mu} c_{n\mu}(\vec{k}) e^{i\vec{k}\cdot(\vec{r}_\mu + \vec{R} - \vec{r})} \phi_\mu(\vec{r} - \vec{r}_\mu - \vec{R}), \tag{8}$$

where $\phi_\mu(\vec{r} - \vec{r}_\mu - \vec{R})$ is a basis set atomic orbital centered on atom μ in the unit cell $\vec{R}$, and $c_{n\mu}(\vec{k})$ are the coefficients of the wave function. The latter must be identical to the ones used for wannierization in M_{mn}. (See the above comment about arbitrary phases.)

Default value: `.false.`

Siesta2Wannier90.UnkGrid1 (*integer*):

Number of points along the first lattice vector in the grid where the periodic part of the wave functions will be plotted.

*Default value:*number of points in the SIESTA mesh along this lattice vector

Siesta2Wannier90.UnkGrid2 (*integer*):

Number of points along the second lattice vector in the grid where the periodic part of the wave functions will be plotted.

*Default value:*number of points in the SIESTA mesh along this lattice vector

Siesta2Wannier90.UnkGrid3 (*integer*):

Number of points along the third lattice vector in the grid where the periodic part of the wave functions will be plotted.

*Default value:*number of points in the SIESTA mesh along this lattice vector

Siesta2Wannier90.UnkGridBinary (*logical*):

Flag that determines whether the periodic part of the wave function in the real space grid is written in binary format (default) or in ASCII format.

Default value: `.true.`

Siesta2Wannier90.NumberOfBands (*integer*):

In spin unpolarized calculations, number of bands that will be initially considered by SIESTA to generate the information required by WANNIER90. Note that it should be at least as large as the index of the highest-lying band in the WANNIER90 post-processing. For example, if the wannierization is going to involve bands 3 to 5, the SIESTA number of bands should be at least 5. Bands 1 and 2 should appear in a "excluded" list.

Default value: Number of occupied bands. To avoid problems of interpretation, it is strongly advised to avoid relying on the defaults and to specify the number of bands explicitly.

Siesta2Wannier90.NumberOfBandsUp (*integer*):

In spin-polarized calculations, number of bands with spin up that will be initially considered by SIESTA to generate the information required by WANNIER90. (See above for details.)

Default value: Number of occupied bands with spin up. To avoid problems of interpretation, it is strongly advised to avoid relying on the defaults and to specify the number of bands explicitly.

Siesta2Wannier90.NumberOfBandsDown (*integer*):

In spin-polarized calculations, number of bands with spin down that will be initially considered by SIESTA to generate the information required by WANNIER90. (See above for details.)

Default value: Number of occupied bands with spin down. To avoid problems of interpretation, it is strongly advised to avoid relying on the defaults and to specify the number of bands explicitly.

6.19 Systems with net charge or dipole, and electric fields

NetCharge (*real*): Specify the net charge of the system (in units of $|e|$). For charged systems, the energy converges very slowly versus cell size. For molecules or atoms, a Madelung correction term is applied to the energy to make it converge much faster with cell size (this is done only if the cell is SC, FCC or BCC). For other cells, or for periodic systems (chains, slabs or bulk), this energy correction term can not be applied, and the user is warned by the program. It is not advised to do charged systems other than atoms and molecules in SC, FCC or BCC cells, unless you know what you are doing.

Use: For example, the F^- ion would have **NetCharge** = `-1`, and the Na^+ ion would have **NetCharge** = `1`. Fractional charges can also be used.

Default value: `0.0`

SimulateDoping (*boolean*):

This option instructs the program to add a background charge density to simulate doping. The new "doping" routine calculates the net charge of the system, and adds a compensating background charge that makes the system neutral. This background charge is constant at points of the mesh near the atoms, and zero at points far from the atoms. This simulates situations like doped slabs, where the extra electrons (holes) are compensated by oposite charges at the material (the ionized dopant impurities), but not at the vacuum. This serves to simulate properly doped systems in which there are large portions of vacuum, such as doped slabs.

(See `Tests/sic-slab`)

Default value: `.false.`

ExternalElectricField (*data block*): It specifies an external electric field for molecules, chains and slabs. The electric field should be orthogonal to 'bulk directions', like those parallel to a slab (bulk electric fields, like in dielectrics or ferroelectrics, are not allowed). If it is not, an error message is issued and the components of the field in bulk directions are suppressed automatically. The input is a vector in Cartesian coordinates, in the specified units. Example:

```
%block ExternalElectricField
   0.000  0.000  0.500  V/Ang
%endblock ExternalElectricField
```

Starting with version 4.0, applying an electric field perpendicular to a slab will by default enable the slab dipole correction option. To reproduce older calculations, set this correction option explicitly to `.false.` in the input file.

Default value: zero field

SlabDipoleCorrection (*boolean*):

If `true`, SIESTA calculates the electric field required to compensate the dipole of the system at every iteration of the self-consistent cycle. The potential added to the grid corresponds to that of a dipole layer at the middle of the vacuum layer. For slabs, this exactly compensates the electric field at the vacuum created by the dipole moment of the system, thus allowing to treat asymmetric slabs (including systems with an adsorbate on one surface) and compute properties such as the work funcion of each of the surfaces.

NOTE: If the program is fed a starting density matrix from an uncorrected calculation (i.e., with an exagerated dipole), the first iteration might use a compensating field that is too big, with the risk of taking the system out of the convergence basin. In that case, it is advisable to use the `DM.MixSCF1` option to request a mix of the input and output density matrices after that first iteration.

(See `Tests/sic-slab`)

This will default to `true` if an external field is applied to a slab calculation, otherwise it will default to `false`.

Default value: `false`

6.20 Output of charge densities and potentials on the grid

SIESTA represents these magnitudes on the real-space grid. The following options control the generation of the appropriate files, which can be processed by the programs in the `Util/Grid` directory, and also by Andrei Postnikov's utilities in `Util/Contrib/APostnikov`. See also `/Util/Denchar` for an alternative way to plot the charge density (and wavefunctions).

SaveRho (*logical*): Instructs to write the valence pseudocharge density at the mesh used by DHSCF, in file `SystemLabel`.RHO.

Use: File `SystemLabel`.RHO is only written, not read, by siesta. This file can be read by routine IORHO, which may be used by other application programs.

If netCDF support is compiled in, the file `Rho.grid.nc` is produced.

Default value: `.false.`

SaveDeltaRho (*logical*): Instructs to write $\delta\rho(\vec{r}) = \rho(\vec{r}) - \rho_{atm}(\vec{r})$, i.e., the valence pseudocharge density minus the sum of atomic valence pseudocharge densities. It is done for the mesh points used by DHSCF and it comes in file `SystemLabel`.DRHO. This file can be read by routine IORHO, which may be used by an application program in later versions.

Use: File `SystemLabel`.DRHO is only written, not read, by siesta.

If netCDF support is compiled in, the file `DeltaRho.grid.nc` is produced.

Default value: `.false.`

SaveRhoXC (*logical*): Instructs to write the valence pseudocharge density at the mesh, including the nonlocal core corrections used to calculate the exchange-correlation energy, in file `SystemLabel`.RHOXC.

Use: File `SystemLabel`.RHOXC is only written, not read, by siesta.

If netCDF support is compiled in, the file `RhoXC.grid.nc` is produced.

Default value: `.false.`

SaveElectrostaticPotential (*logical*): Instructs to write the total electrostatic potential, defined as the sum of the hartree potential plus the local pseudopotential, at the mesh used by DHSCF, in file `SystemLabel`.VH. This file can be read by routine IORHO, which may be used by an application program in later versions.

Use: File `SystemLabel`.VH is only written, not read, by siesta.

If netCDF support is compiled in, the file `ElectrostaticPotential.grid.nc` is produced.

Default value: `.false.`

SaveNeutralAtomPotential (*logical*): Instructs to write the neutral-atom potential, defined as the sum of the hartree potential of a "pseudo atomic valence charge" plus the local pseudopotential, at the mesh used by DHSCF, in file `SystemLabel`.VNA. It is written at the start of the self-consistency cycle, as this potential does not change.

Use: File `SystemLabel`.VNA is only written, not read, by siesta.

If netCDF support is compiled in, the file `Vna.grid.nc` is produced.

Default value: `.false.`

SaveTotalPotential (*logical*): Instructs to write the valence total effective local potential (local pseudopotential + Hartree + Vxc), at the mesh used by DHSCF, in file `SystemLabel`.VT. This file can be read by routine IORHO, which may be used by an application program in later versions.

Use: File `SystemLabel`.VT is only written, not read, by siesta.

If netCDF support is compiled in, the file `TotalPotential.grid.nc` is produced.

Default value: `.false.`

Side effect: the vacuum level, defined as the effective potential at grid points with zero density, is printed in the standard output whenever such points exist (molecules, slabs) and either SaveElectrostaticPotential or SaveTotalPotential are true. In a symetric (nonpolar) slab, the work function can be computed as the difference between the vacuum level and the Fermi energy.

SaveIonicCharge (*logical*): Instructs to write the soft diffuse ionic charge at the mesh used by DHSCF, in file `SystemLabel`.IOCH. This file can be read by routine IORHO, which may be used by an application program in later versions. Remember that, within the SIESTA sign convention, the electron charge density is positive and the ionic charge density is negative.

Use: File `SystemLabel`.IOCH is only written, not read, by siesta.

If netCDF support is compiled in, the file `Chlocal.grid.nc` is produced.

Default value: `.false.`

SaveTotalCharge (*logical*): Instructs to write the total charge density (ionic+electronic) at the mesh used by DHSCF, in file `SystemLabel`.TOCH. This file can be read by routine IORHO, which may be used by an application program in later versions. Remember that, within the SIESTA sign convention, the electron charge density is positive and the ionic charge density is negative.

Use: File `SystemLabel`.TOCH is only written, not read, by siesta.

Default value: `.false.`

SaveBaderCharge (*logical*): Instructs the program to save the charge density for further post-processing by a Bader-analysis program. This "Bader charge" is the sum of the electronic valence charge density and a set of "model core charges" placed at the atomic sites. For a given atom, the model core charge is a generalized Gaussian, but confined to a radius of 1.0 Bohr (by default), and integrating to the total core charge (Z-$Z_{\rm val}$). These core charges are needed to provide local maxima for the charge density at the atomic sites, which are not guaranteed in a pseudopotential calculation. For hydrogen, an artificial core of 1 electron is added, with a confinement radius of 0.6 Bohr by default. The Bader charge is projected on the grid points of the mesh used by DHSCF, and saved in file

`SystemLabel`.BADER. This file can be post-processed by the program `Util/grid2cube` to convert it to the "cube" format, accepted by several Bader-analysis programs (for example, see `http://theory.cm.utexas.edu/bader/`). Due to the need to represent a localized core charge, it is advisable to use a moderately high MeshCutoff when invoking this option (300-500 Ry). The size of the "basin of attraction" around each atom in the Bader analysis should be monitored to check that the model core charge is contained in it.

The radii for the model core charges can be specified in the input fdf file. For example:

```
bader-core-radius-standard 1.3 Bohr
```

```
bader-core-radius-hydrogen 0.4 Bohr
```

The suggested way to run the Bader analysis with the Univ. of Texas code is to use both the RHO and BADER files (both in "cube" format), with the BADER file providing the "reference" and the RHO file the actual significant valence charge data which is important in bonding. (See the notes for pseudopotential codes in the above web page.) For example, for the h2o-pop example:

```
bader h2o-pop.RHO.cube -ref h2o-pop.BADER.cube
```

If netCDF support is compiled in, the file `BaderCharge.grid.nc` is produced.

Default value: `.false.`

AnalyzeChargeDensityOnly (*logical*):

If "true", the program optionally generates charge density files and computes partial atomic charges (Hirshfeld, Voronoi, Bader) from the information in the input density matrix, and stops. This is useful to analyze the properties of the charge density without a diagonalization step, and with a user-selectable mesh cutoff. Note that the **DM.UseSaveDM** option should be active. Note also that if an initial density matrix (DM file) is used, it is not normalized. All the relevant fdf options for charge-density file production and partial charge calculation can be used with this option.

Default value: `.false.`

SaveInitialChargeDensity (*logical*):

If "true", the program generates a `SystemLabel`.RHOINIT file (and a `RhoInit.grid.nc` file if netCDF support is compiled in) containing the charge density used to start the first self-consistency step, and it stops. Note that if an initial density matrix (DM file) is used, it is not normalized. This is useful to generate the charge density associated to "partial" DMs, as created by progras such as `dm_creator` and `dm_filter`.

(This option is to be deprecated in favor of **AnalyzeChargeDensityOnly**).

Default value: `.false.`

6.21 Auxiliary Force field

It is possible to supplement the DFT interactions with a limited set of force-field options, typically useful to simulate dispersion interactions. It is not yet possible to turn off DFT and

base the dynamics only on the force field. The `GULP` program should be used for that.

MM.Potentials (*data block*): This block allows the input of molecular mechanics potentials between species. The following potentials are currently implemented:

- C6, C8, C10 powers of the Tang-Toennes damped dispersion potential.
- A harmonic interaction.
- A dispersion potential of the Grimme type (similar to the C6 type but with a different damping function). (See S. Grimme, J. Comput. Chem. Vol 27, 1787-1799 (2006)). See also `MM.Grimme.D` and `MM.Grimme.S6` below.

The format of the input is the two species numbers that are to interact, the potential name (C6, C8, C10, harm, or Grimme), followed by the potential parameters. For the damped dispersion potentials the first number is the coefficient and the second is the exponent of the damping term (i.e., a reciprocal length). A value of zero for the latter term implies no damping. For the harmonic potential the force constant is given first, followed by r0. For the Grimme potential C6 is given first, followed by the (corrected) sum of the van der Waals radii for the interacting species (a real length). Positive values of the C6, C8, and C10 coefficients imply attractive potentials.

Use: Gives the input for the molecular mechanics potentials.

```
%block MM.Potentials
   1 1 C6 32.0 2.0
   1 2 harm 3.0 1.4
   2 3 Grimme 6.0 3.2
%endblock MM.Potentials
```

Default value: None.

MM.Cutoff (*physical*): Specifies the distance out to which molecular mechanics potential will act before being treated as going to zero.

Use: Limits the real space range of the molecular mechanics potentials.

Default value: `30.0 Bohr`

MM.UnitsEnergy (*units*): Specifies the units to be used for energy in the molecular mechanics potentials.

Use: Controls the units for energy in the molecular mechanics input.

Default value: `eV`

MM.UnitsDistance (*units*): Specifies the units to be used for distance in the molecular mechanics potentials.

Use: Controls the units for distance in the molecular mechanics input.

Default value: `Ang`

MM.Grimme.D : Specifies the scale factor d for the scaling function in the Grimme dispersion potential (see above).

Default value: 20.0

MM.Grimme.S6 : Specifies the overall fitting factor s_6 for the Grimme dispersion potential (see above). This number depends on the quality of the basis set, the exchange-correlation functional, and the fitting set.

Default value: 1.66 (for DZP basis sets).

6.22 Parallel options

BlockSize (*integer*): The orbitals are distributed over the processors when running in parallel using a 1-D block-cyclic algorithm. **BlockSize** is the number of consecutive orbitals which are located on a given processor before moving to the next one. Large values of this parameter lead to poor load balancing, while small values can lead to inefficient execution. The performance of the parallel code can be optimised by varying this parameter until a suitable value is found.

Use: Controls the blocksize used for distributing orbitals over processors

Default value: Estimated by a heuristic algorithm, with a maximum value of 24. The imbalance in the distribution is not larger than 2.

ProcessorY (*integer*): The mesh points are divided in the Y and Z directions (more precisely, along the second and third lattice vectors) over the processors in a 2-D grid. **ProcessorY** specifies the dimension of the processor grid in the Y-direction and must be a factor of the total number of processors. Ideally the processors should be divided so that the number of mesh points per processor along each axis is as similar as possible.

Use: Controls the dimensions of the 2-D processor grid for mesh distribution

Default value: Variable - chosen using multiples of factors of the total number of processors

Diag.Memory (*real no units*): Whether the parallel diagonalisation of a matrix is successful or not can depend on how much workspace is available to the routine when there are clusters of eigenvalues. **Diag.Memory** allows the user to increase the memory available, when necessary, to achieve successful diagonalisation and is a scale factor relative to the minimum amount of memory that SCALAPACK might need.

Use: Controls the amount of workspace available to parallel matrix diagonalisation

Default value: 1.0

Diag.ParallelOverK (*logical*): For the diagonalisation there is a choice in strategy about whether to parallelise over the K points or over the orbitals. K point diagonalisation is close to perfectly parallel but is only useful where the number of K points is much larger than the number of processors and therefore orbital parallelisation is generally preferred.

The exception is for metals where the unit cell is small, but the number of K points to be sampled is very large. In this last case it is recommend that this option be used.

NOTE: This scheme is not used for the diagonalizations involved in the generation of the band-structure (as specified with **BandLines** or **BandPoints**) or in the generation of wave-function information (as specified with **WaveFuncKpoints**). In these cases the program falls back to using parallelization over orbitals.

Use: Controls whether the diagonalisation is parallelised with respect to orbitals or K points - not allowed for non-co-linear spin case.

Default value: false

6.22.1 Parallel decompositions for O(N)

Apart from the default block-cyclic decomposition of the orbital data, O(N) calculations can use other schemes which should be more efficient: spatial decomposition (based on atom proximity), and domain decomposition (based on the most efficient abstract partition of the interaction graph of the Hamiltonian).

UseDomainDecomposition (*logical*): This option instructs the program to employ a graph-partitioning algorithm (using the `METIS` library (See `www.cs.umn.edu/~metis`) to find an efficient distribution of the orbital data over processors. To use this option (meaningful only in parallel) the program has to be compiled with the preprocessor option `ON_DOMAIN_DECOMP` and the `METIS` library has to be linked in.

Default value: false

UseSpatialDecomposition (*logical*): When performing a parallel order N calculation, this option instructs the program to execute a spatial decomposition algorithm in which the system is divided into cells, which are then assigned, together with the orbitals centered in them, to the different processors. The size of the cells is, by default, equal to the maximum distance at which there is a non-zero matrix element in the Hamiltonian between two orbitals, or the radius of the Localized Wannier function - which ever is the larger. If this is the case, then an orbital will only interact with other orbitals in the same or neighbouring cells. However, by decreasing the cell size and searching over more cells it is possible to achieve better load balance in some cases. This is controlled by the variable **RcSpatial**.

NOTE: The distribution algorithm is quite fragile and a careful tuning of **RcSpatial** might be needed. This option is therefore not enabled by default.

Default value: false

RcSpatial (*real distance*):

Controls the cell size during the spatial decomposition.

Default value: maximum of the matrix element range or the Localized Wannier Function radius

6.23 Efficiency options

DirectPhi (*logical*): The calculation of the matrix elements on the mesh requires the value of the orbitals on the mesh points. This array represents one of the largest uses of memory within the code. If set to true this option allows the code to generate the orbital values when needed rather than storing the values. This obviously costs more computer time but will make it possible to run larger jobs where memory is the limiting factor.

Use: Controls whether the values of the orbitals at the mesh points are stored or calculated on the fly.

Default value: false

6.24 Memory, CPU-time, and Wall time accounting options

AllocReportLevel (*integer*): Sets the level of the allocation report, printed in file `SystemLabel`.alloc. However, not all the allocated arrays are included in the report (this will be corrected in future versions). The allowed values are:

- level 0 : no report at all (the default)
- level 1 : only total memory peak and where it occurred
- level 2 : detailed report printed only at normal program termination
- level 3 : detailed report printed at every new memory peak
- level 4 : print every individual (re)allocation or deallocation

NOTE: In MPI runs, only node-0 peak reports are produced.

AllocReportThreshold (*real*): Sets the minimum size (in bytes) of the arrays whose memory use is individually printed in the detailed allocation reports (levels 2 and 3). It does not affect the reported memory sums and peaks, which always include all arrays.

Default value: `0.0`

TimerReportThreshold (*real*): Sets the minimum fraction, of total CPU time, of the subroutines or code sections whose CPU time is individually printed in the detailed timer reports. To obtain the accounting of MPI communication times in parallel executions, you must compile with option `-DMPI_TIMING`. In serial execution, the CPU times are printed at the end of the output file. In parallel execution, they are reported in a separated file named `SystemLabel`.times.

Default value: `0.0`

UseTreeTimer (*logical*):

Enable an experimental timer which is based on wall time on the master node and is aware of the tree-structure of the timed sections. At the end of the program, a report is generated in the output file, and a `time.json` file in JSON format is also written. This file can be used by third-party scripts to process timing data.

Default value: `.false.`

MaxWalltime (*real time*):

Set an internal limit to the wall time allotted to the program's execution. Typically this is related to the external limit imposed by queuing systems. The code checks its wall time periodically and will abort if nearing the limit, with some slack left for clean-up operations (proper closing of files, emergency output...), as determined by **MaxWalltime.Slack**. See Sec. 14 for available units of time (**s**, **mins**, **hours**, **days**).

Default value: `Infinity`

MaxWalltime.Slack (*real time*):

The code checks its wall time $T_{\rm wall}$ periodically and will abort if $T_{\rm wall} > T_{\rm max} - T_{\rm slack}$, so that some slack is left for any clean-up operations.

Default value: `5 seconds`

6.25 The catch-all option UseSaveData

This is a dangerous feature, and is deprecated, but retained for historical compatibility. Use the individual options instead.

UseSaveData (*logical*): Instructs to use as much information as possible stored from previous runs in files `SystemLabel`.XV, `SystemLabel`.DM and `SystemLabel`.LWF, where SystemLabel is the name associated to parameter `SystemLabel`.

Use: If the required files do not exist, warnings are printed but the program does not stop.

Default value: `.false.`

6.26 Output of information for Denchar

The program `denchar` in `Util/Denchar` can generate charge-density and wavefunction information in real space.

WriteDenchar (*logical*): Instructs to write information needed by the utility program DENCHAR (by J. Junquera and P. Ordejón) to generate valence charge densities and/or wavefunctions in real space (see `Util/Denchar`). The information is written in files `SystemLabel`.PLD and `SystemLabel`.DIM.

To run DENCHAR you will need, apart from the PLD and DIM files, the Density-Matrix (DM) file and/or a wavefunction (WFSX) file, and the .ion files containing the information about the basis orbital.s

Default value: `.false.`

7 STRUCTURAL RELAXATION, PHONONS, AND MOLECULAR DYNAMICS

This functionality is not SIESTA-specific, but is implemented to provide a more complete simulation package. The program has an outer geometry loop: it computes the electronic structure (and thus the forces and stresses) for a given geometry, updates the atomic positions (and maybe the cell vectors) accordingly and moves on to the next cycle.

Several options for MD and structural optimizations are implemented, selected by **MD.TypeOfRun** (*string*):

- `CG` Coordinate optimization by conjugate gradients). Optionally (see variable MD.VariableCell below), the optimization can include the cell vectors.
- `Broyden` Coordinate optimization by a modified Broyden scheme). Optionally, (see variable MD.VariableCell below), the optimization can include the cell vectors.
- `FIRE` Coordinate optimization by Fast Inertial Relaxation Engine (FIRE) (E. Bitzek et al, PRL 97, 170201, (2006)). Optionally, (see variable MD.VariableCell below), the optimization can include the cell vectors.
- `Verlet` Standard Verlet algorithm MD
- `Nose` MD with temperature controlled by means of a Nosé thermostat
- `ParrinelloRahman` MD with pressure controlled by the Parrinello-Rahman method
- `NoseParrinelloRahman` MD with temperature controlled by means of a Nosé thermostat and pressure controlled by the Parrinello-Rahman method
- `Anneal` MD with annealing to a desired temperature and/or pressure (see variable MD.AnnealOption below)
- `FC` Compute force constants matrix for phonon calculations.
- `Forces` (Receive coordinates from, and return forces to, an external driver program, using MPI, Unix pipes, or Inet sockets for communication. The routines in module fsiesta allow the user's program to perform this communication transparently, as if siesta were a conventional force-field subroutine. See `Util/SiestaSubroutine/README` for details. WARNING: if this option is specified without a driver program sending data, siesta may hang without any notice).

  See directory Util/Scripting for other driving options.

Default value: `CG` (except if **compat-pre-v4-dynamics** is set. See the notes below.

Note that some options specified in later variables (like quenching) modify the behavior of these MD options.

Appart from being able to act as a force subroutine for a driver program that uses module fsiesta, SIESTA is also prepared to communicate with the i-PI code (see

http://epfl-cosmo.github.io/gle4md/index.html?page=ipi). To do this, SIESTA must be started after i-PI (it acts as a client of i-PI, communicating with it through Inet or Unix sockets), and the following lines must be present in the .fdf data file:

```
MD.TypeOfRun      Master     # equivalent to 'Forces'
Master.code       i-pi       # ( fsiesta | i-pi )
Master.interface  socket     # ( pipes | socket | mpi )
Master.address    localhost  # or driver's IP, e.g. 150.242.7.140
Master.port       10001      # 10000+siesta_process_order
Master.socketType inet       # ( inet | unix )
```

7.1 Compatibility with pre-v4 versions

Starting in the summer of 2015, some changes were made to the behavior of the program regarding default dynamics options and choice of coordinates to work with during post-processing of the electronic structure. The changes are:

- The default dynamics option is "CG" instead of "Verlet".
- The coordinates, if moved by the dynamics routines, are reset to their values at the previous step for the analysis of the electronic structure (band structure calculations, DOS, LDOS, etc).
- Some output files reflect the values of the "un-moved" coordinates.

To recover the previous behavior, the user can turn on the compatibility switch **compat-pre-v4-dynamics**, which is off by default.

Note that complete compatibility cannot be perfectly guaranteed.

7.2 Structural relaxation

In this mode of operation, the program moves the atoms (and optionally the cell vectors) trying to minimize the forces (and stresses) on them.

These are the options common to all relaxation methods. If the Zmatrix input option is in effect (see Sec. 6.4.2) the Zmatrix-specific options take precedence. The 'MD' prefix is misleading but kept for historical reasons.

MD.VariableCell (*logical*): If true, the lattice is relaxed together with the atomic coordinates. It allows to target hydrostatic pressures or arbitrary stress tensors. See **MD.MaxStressTol**, **MD.TargetPressure**, **MD.TargetStress**, **MD.ConstantVolume**, and **MD.PreconditionVariableCell**.

Use: Used only if MD.TypeOfRun is `CG` or `Broyden` or `FIRE`

Default value: `.false.`

MD.ConstantVolume (*logical*): If true, the cell volume is kept constant in a variable-cell relaxation: only the cell shape and the atomic coordinates are allowed to change. Note that it does not make much sense to specify a target stress or pressure in this case, except for anisotropic (traceless) stresses. See **MD.VariableCell**, **MD.TargetStress**.

Use: Used only if MD.TypeOfRun is `CG` or `Broyden` or `FIRE`, and MD.VariableCell is `.true.`.

Default value: `.false.`

MD.RelaxCellOnly (*logical*):

If true, only the cell parameters are relaxed (by the Broyden or FIRE method, not CG). The atomic coordinates are re-scaled to the new cell, keeping the fractional coordinates constant. For Zmatrix calculations, the fractional position of the first atom in each molecule is kept fixed, and no attempt is made to rescale the bond distances or angles.

Use: Used only if MD.TypeOfRun is `FIRE` or `Broyden` and MD.VariableCell is `.true.`.

Default value: `.false.`

MD.MaxForceTol (*real force*): Force tolerance in coordinate optimization. Run stops if the maximum atomic force is smaller than **MD.MaxForceTol** (see **MD.MaxStressTol** for variable cell).

Use: Used only if MD.TypeOfRun is `CG` or `Broyden` or `FIRE`

Default value: `0.04 eV/Ang`

MD.MaxStressTol (*real pressure*): Stress tolerance in variable-cell CG optimization. Run stops if the maximum atomic force is smaller than **MD.MaxForceTol** and the maximum stress component is smaller than **MD.MaxStressTol**.

Use: Used only if MD.TypeOfRun is `CG` or `Broyden` or `FIRE`, and Md.VariableCell is `.true.`

Special consideration is needed if used with Sankey-type basis sets, since the combination of orbital kinks at the cutoff radii and the finite-grid integration originate discontinuities in the stress components, whose magnitude depends on the cutoff radii (or energy shift) and the mesh cutoff. The tolerance has to be larger than the discontinuities to avoid endless optimizations if the target stress happens to be in a discontinuity.

Default value: `1.0 GPa`

MD.NumCGsteps (*integer*): Maximum number of conjugate gradient (or Broyden) minimization moves (the minimization will stop if tolerance is reached before; see MD.MaxForceTol below).

Use: Used only if MD.TypeOfRun is `CG` or `Broyden`

Default value: `0`

MD.MaxCGDispl (*real length*): Maximum atomic displacements in an optimization move.

Use: Used only if MD.TypeOfRun is `CG` or `Broyden` or `FIRE` (despite its name). For the Broyden optimization method, it is also possible to limit indirectly the *initial* atomic

displacements using **MD.Broyden.Initial.Inverse.Jacobian**. For the FIRE method, the same result can be obtained by choosing a small time step.

Note that there are Zmatrix-specific options that override this option.

Default value: `0.2 Bohr`

MD.PreconditionVariableCell (*real length*): A length to multiply to the strain components in a variable-cell optimization. The strain components enter the minimization on the same footing as the coordinates. For good efficiency, this length should make the scale of energy variation with strain similar to the one due to atomic displacements. It is also used for the application of the **MD.MaxCGDispl** value to the strain components.

Use: Used only if MD.TypeOfRun is `CG` or `Broyden` or `FIRE` and MD.VariableCell is `.true.`

Default value: `5.0 Ang`

ZM.ForceTolLength (*real force*): Parameter that controls the convergence with respect to forces on Z-matrix lengths

Use: This option sets the convergence criteria for the forces that act on Z-matrix components with units of length.

Default value: 0.00155574 `Ry/Bohr`

ZM.ForceTolAngle (*torque*): Parameter that controls the convergence with respect to forces on Z-matrix angles

Use: This option sets the convergence criteria for the forces that act on Z-matrix components with units of angle.

Default value: 0.00356549 `Ry/rad`

ZM.MaxDisplLength (*real length*): Parameter that controls the maximum change in a Z-matrix length during an optimisation step.

Use: This option sets the maximum displacement for a Z-matrix length

Default value: `0.2 Bohr`

ZM.MaxDisplAngle (*real angle*): Parameter that controls the maximum change in a Z-matrix angle during an optimisation step.

Use: This option sets the maximum displacement for a Z-matrix angle

Default value: `0.003 rad`

7.2.1 Conjugate-gradients optimization

This was historically the default geometry-optimization method, and all the above options were introduced specifically for it, hence their names. The following pertains only to this method:

MD.UseSaveCG (*logical*): Instructs to read the conjugate-gradient hystory information stored in file `SystemLabel`.CG by a previous run.

Use: To get actual continuation of iterrupted CG runs, use together with **MD.UseSaveXV** = `.true.` with the XV file generated in the same run as the CG file. If the required file does not exist, a warning is printed but the program does not stop. Overrides **UseSaveData**.

Default value: `.false.`

(No such feature exists yet for a Broyden-based relaxation.)

7.2.2 Broyden optimization

It uses the modified Broyden algorithm to build up the Jacobian matrix. (See D.D. Johnson, PRB 38, 12807 (1988)). (Note: This is not BFGS.)

MD.Broyden.History.Steps (*integer*):

Number of relaxation steps during which the modified Broyden algorithm builds up the Jacobian matrix.

Use: Used only if MD.TypeOfRun is `Broyden`.

Default value: `5`

MD.Broyden.Cycle.On.Maxit (*logical*):

Upon reaching the maximum number of history data sets which are kept for Jacobian estimation, throw away the oldest and shift the rest to make room for a new data set. The alternative is to re-start the Broyden minimization algorithm from a first step of a diagonal inverse Jacobian (which might be useful when the minimization is stuck).

Use: Used only if MD.TypeOfRun is `Broyden`.

Default value: `.true.`

MD.Broyden.Initial.Inverse.Jacobian (*real*):

Initial inverse Jacobian for the optimization procedure. (The units are those implied by the internal Siesta usage (Bohr for lenghts and Ry for energies). The default value seems to work well for most systems.

Use: Used only if MD.TypeOfRun is `Broyden`.

Default value: `1.0`

7.2.3 FIRE relaxation

Implementation of the Fast Inertial Relaxation Engine (FIRE) method (E. Bitzek et al, PRL 97, 170201, (2006) in a manner compatible with the CG and Broyden modes of relaxation. (An older implementation activated by the **MD.FireQuench** variable is still available).

MD.FIRE.TimeStep (*real time*):

The (fictitious) time-step for FIRE relaxation. This is the main user-variable when the option `FIRE` for `MD.TypeOfRun` is active.

Default value: The molecular-dynamics time-step, as specified by `MD.LengthTimeStep`, but this is misleading and should be avoided.

There are other low-level options tunable by the user (see the routines `fire_optim` and `cell_fire_optim` for more details.

7.2.4 Quenched MD

These methods are really based on molecular dynamics, but are used for structural relaxation.

Note that the Zmatrix input option (see Sec. 6.4.2) is not compatible with molecular dynamics. The initial geometry can be specified using the Zmatrix format, but the Zmatrix generalized coordinates will not be updated.

Note also that the force and stress tolerances have no effect on the termination conditions of these methods. They run for the number of MD steps requested (this is arguably a bug).

MD.Quench (*logical*): Logical option to perform a power quench during the molecular dynamics. In the power quench, each velocity component is set to zero if it is opposite to the corresponding force of that component. This affects atomic velocities, or unit-cell velocities (for cell shape optimizations).

Use: Used only if **MD.TypeOfRun** = `Verlet` or `ParrinelloRahman`. It is incompatible with Nose thermostat options. The quench option allows structural relaxations of only atomic coordinates (with **MD.TypeOfRun** = `Verlet`) or atomic coordinates AND cell shape (with **MD.TypeOfRun** = `ParrinelloRahman`). **MD.Quench** is superseded by **MD.FireQuench** (see below).

Default value: `.false.`

MD.FireQuench (*logical*) (Deprecated)

SEE the new option `FIRE` for `MD.TypeOfRun`

Logical option to perform a FIRE quench during a Verlet molecular dynamics run, as described by Bitzek *et al.* in Phys. Rev. Lett. **97**, 170201 (2006). It is a relaxation algorithm, and thus the dynamics are of no interest per se: the initial time-step can be played with (it uses **MD.LengthTimeStep** as initial Δt), as well as the initial temperature (recommended 0) and the atomic masses (recommended equal). Preliminary tests seem to indicate that the combination of $\Delta t = 5$ fs and a value of 20 for the atomic masses works reasonably. The dynamics stops when the force tolerance is reached (**MD.MaxForceTol**). The other parameters controlling the algorithm (initial damping, increase and decrease thereof etc.) are hardwired in the code, at the recommended values in the cited paper, including $\Delta t_{max} = 10$ fs.

Use: Used only if **MD.TypeOfRun** = `Verlet`. It is incompatible with Nose thermostat options. No variable cell option implemented for this at this stage. **MD.FireQuench** supersedes **MD.Quench**. This option is deprecated. The new option `FIRE` for `MD.TypeOfRun` should be used instead.

Default value: `.false.`

7.3 Target stress options

Useful for structural optimizations and constant-pressure molecular dynamics.

MD.TargetPressure (*real pressure*): Target pressure for Parrinello-Rahman method, variable cell optimizations, and annealing options.

Use: Used only if MD.TypeOfRun = `ParrinelloRahman`, `NoseParrinelloRahman`, `CG, Broyden, or FIRE` (variable cell), or `Anneal` (if MD.AnnealOption = `Pressure` or `TemperatureandPressure`)

Default value: `0.0 GPa`

MD.TargetStress (*data block*): External or target stress tensor for variable cell optimizations. Stress components are given in a line, in the order `xx, yy, zz, xy, xz, yz`. In units of **MD.TargetPressure**, but with the opposite sign. For example, a uniaxial compressive stress of 2 GPa along the 100 direction would be given by

```
MD.TargetPressure  2. GPa
%block MD.TargetStress
    -1.0  0.0  0.0  0.0  0.0  0.0
%endblock MD.TargetStress
```

Use: Used only if MD.TypeOfRun is `CG, Broyden, or FIRE` and MD.VariableCell is `.true.`

Default value: Hydrostatic target pressure: `-1., -1., -1., 0., 0., 0.`

MD.RemoveIntramolecularPressure (*logical*):

If `.true.`, the contribution to the stress coming from the internal degrees of freedom of the molecules will be subtracted from the stress tensor used in variable-cell optimization or variable-cell molecular-dynamics. This is done in an approximate manner, using the virial form of the stress, and assumming that the "mean force" over the coordinates of the molecule represents the "inter-molecular" stress. The correction term was already computed in earlier versions of Siesta and used to report the "molecule pressure". The correction is now computed molecule-by-molecule if the Zmatrix format is used.

If the intra-molecular stress is removed, the corrected static and total stresses are printed in addition to the uncorrected items. The corrected Voigt form is also printed.

Default value: `.false.`

7.4 Molecular dynamics

In this mode of operation, the program moves the atoms (and optionally the cell vectors) in response to the forces (and stresses), using the classical equations of motion.

Note that the Zmatrix input option (see Sec. 6.4.2) is not compatible with molecular dynamics. The initial geometry can be specified using the Zmatrix format, but the Zmatrix generalized coordinates will not be updated.

MD.InitialTimeStep (*integer*): Initial time step of the MD simulation. In the current version of SIESTA it must be 1.

Use: Used only if MD.TypeOfRun is not `CG` or `Broyden`

Default value: `1`

MD.FinalTimeStep (*integer*): Final time step of the MD simulation.

Default value: `1`

MD.LengthTimeStep (*real time*): Length of the time step of the MD simulation.

Default value: `1.0 fs`

MD.InitialTemperature (*real temperature or energy*): Initial temperature for the MD run. The atoms are assigned random velocities drawn from the Maxwell-Bolzmann distribution with the corresponding temperature. The constraint of zero center of mass velocity is imposed.

Use: Used only if **MD.TypeOfRun** = `Verlet, Nose, ParrinelloRahman, NoseParrinelloRahman` or `Anneal`.

Default value: `0.0 K`

MD.TargetTemperature (*real temperature or energy*): Target temperature for Nose thermostat and annealing options.

Use: Used only if **MD.TypeOfRun** = `Nose, NoseParrinelloRahman` or `Anneal` (if **MD.AnnealOption** = `Temperature` or `TemperatureandPressure`)

Default value: `0.0 K`

MD.NoseMass (*real moment of inertia*): Generalized mass of Nose variable. This determines the time scale of the Nose variable dynamics, and the coupling of the thermal bath to the physical system.

Use: Used only if **MD.TypeOfRun** = `Nose` or `NoseParrinelloRahman`

Default value: `100.0 Ry*fs**2`

MD.ParrinelloRahmanMass (*real moment of inertia*): Generalized mass of Parrinello-Rahman variable. This determines the time scale of the Parrinello-Rahman variable dynamics, and its coupling to the physical system.

Use: Used only if **MD.TypeOfRun** = `ParrinelloRahman` or `NoseParrinelloRahman`

Default value: `100.0 Ry*fs**2`

Default value: Same as **NumberOfAtoms**

MD.AnnealOption (*string*): Type of annealing MD to perform. The target temperature or pressure are achieved by velocity and unit cell rescaling, in a given time determined by the variable **MD.TauRelax** below.

- `Temperature` (Reach a target temperature by velocity rescaling)
- `Pressure` (Reach a target pressure by scaling of the unit cell size and shape)

- `TemperatureandPressure` (Reach a target temperature and pressure by velocity rescaling and by scaling of the unit cell size and shape)

Use: Used only if **MD.TypeOfRun** = `Anneal`

Default value: `TemperatureAndPressure`

MD.TauRelax (*real time*): Relaxation time to reach target temperature and/or pressure in annealing MD. Note that this is a "relaxation time", and as such it gives a rough estimate of the time needed to achieve the given targets. As a normal simulation also exhibits oscillations, the actual time needed to reach the *averaged* targets will be significantly longer.

Use: Used only if **MD.TypeOfRun** = `Anneal`

Default value: `100.0 fs`

MD.BulkModulus (*real pressure*): Estimate (may be rough) of the bulk modulus of the system. This is needed to set the rate of change of cell shape to reach target pressure in annealing MD.

Use: Used only if **MD.TypeOfRun** = `Anneal`, when **MD.AnnealOption** = `Pressure` or `TemperatureAndPressure`

Default value: `100.0 Ry/Bohr**3`

7.5 Output options for dynamics

Every time the atoms move, either during coordinate relaxation or molecular dynamics, their positions **predicted for next step** and **current** velocities are stored in file SystemLabel.XV. The shape of the unit cell and its associated 'velocity' (in Parrinello-Rahman dynamics) are also stored in this file.

Other options follow.

WriteCoorInitial (*logical*): It determines whether the initial atomic coordinates of the simulation are dumped into the main output file. These coordinates correspond to the ones actually used in the first step (see the section on precedence issues in structural input) and are output in Cartesian coordinates in Bohr units.

It is not affected by the setting of **LongOutput**.

Default value: `.true.`

WriteCoorStep (*logical*): If `.true.` it writes the atomic coordinates to standard output at every MD time step or relaxation step. The coordinates are always written in the *Systemlabel*.XV file, but overriden at every step. They can be also accumulated in the *Systemlabel*.MD or *Systemlabel*.MDX files depending on **WriteMDhistory**.

Default value: `.false.` (see **LongOutput**)

WriteForces (*logical*): If `.true.` it writes the atomic forces to the output file at every MD time step or relaxation step. Note that the forces of the last step can be found in the file *Systemlabel*.FA. If constraints are used, the file *SystemLabel*.`FAC` is also written.

Default value: `.false.` (see **LongOutput**)

WriteMDhistory (*logical*): If `.true.` SIESTA accumulates the molecular dynamics trajectory in the following files:

- *Systemlabel*.MD : atomic coordinates and velocities (and lattice vectors and their time derivatives, if the dynamics implies variable cell). The information is stored unformatted for postprocessing with utility programs to analyze the MD trajectory.
- *Systemlabel*.MDE : shorter description of the run, with energy, temperature, etc., per time step.

These files are accumulative even for different runs.

Default value: `.false.`

The trajectory of a molecular dynamics run (or a conjugate gradient minimization) can be accumulated in different files: SystemLabel.MD, SystemLabel.MDE, and SystemLabel.ANI. The first file keeps the whole trajectory information, meaning positions and velocities at every time step, including lattice vectors if the cell varies. NOTE that the positions (and maybe the cell vectors) stored at each time step are the **predicted** values for the next step. Care should be taken if joint position-velocity correlations need to be computed from this file. The second gives global information (energy, temperature, etc), and the third has the coordinates in a form suited for XMol animation. See the **WriteMDhistory** and **WriteMDXmol** data descriptors above for information. SIESTA always appends new information on these files, making them accumulative even for different runs.

The `iomd` subroutine can generate both an unformatted file SystemLabel.MD (default) or ASCII formatted files SystemLabel.MDX and SystemLabel.MDC containing the atomic and lattice trajectories, respectively. Edit the file to change the settings if desired.

7.6 Restarting geometry optimizations and MD runs

Every time the atoms move, either during coordinate relaxation or molecular dynamics, their **positions predicted for next step** and **current velocities** are stored in file SystemLabel.XV, where SystemLabel is the value of that FDF descriptor (or 'siesta' by default). The shape of the unit cell and its associated 'velocity' (in Parrinello-Rahman dynamics) are also stored in this file. For MD runs of type Verlet, Parrinello-Rahman, Nose, Nose-Parrinello-Rahman, or Anneal, a file named SystemLabel.VERLET_RESTART, SystemLabel.PR_RESTART, SystemLabel.NOSE_RESTART, SystemLabel.NPR_RESTART, or SystemLabel.ANNEAL_RESTART, respectively, is created to hold the values of auxiliary variables needed for a completely seamless continuation.

If the restart file is not available, a simulation can still make use of the XV information, and "restart" by basically repeating the last-computed step (the positions are shifted backwards by using a single Euler-like step with the current velocities as derivatives). While this feature does not result in seamless continuations, it allows cross-restarts (those in which a simulation of one kind (e.g., Anneal) is followed by another (e.g., Nose)), and permits to re-use dynamical information from old runs.

This restart fix is not satisfactory from a fundamental point of view, so the MD subsystem in Siesta will have to be redesigned eventually. In the meantime, users are reminded that the scripting hooks being steadily introduced (see Util/Scripting) might be used to create custom-made MD scripts.

7.7 Use of general constraints

Note: The Zmatrix format (see Sec. 6.4.2) provides an alternative constraint formulation which can be useful for system involving molecules.

GeometryConstraints (*data block*) Fixes constraints to the change of atomic coordinates during geometry relaxation or molecular dynamics. Allowed constraints are:

- `cellside`: fixes the unit-cell side lengths to their initial values (not implemented yet).
- `cellangle`: fixes the unit-cell angles to their initial values (not implemented yet).
- `stress`: fixes the specified stresses to their initial values.
- `position`: fixes the positions of the specified atoms to their initial values.
- `center`: fixes the center (mean position, not center of mass) of a group of atoms to its initial value (not implemented yet).
- `rigid`: fixes the relative positions of a group of atoms, without restricting their displacement or rotation as a rigid unit (not implemented yet).
- `routine`: Additionally, the user may write a problem-specific routine called **constr** (with the same interface as in the example below), which inputs the atomic forces and stress tensor and outputs them orthogonalized to the constraints. For example, to maintain the relative height of atoms 1 and 2:

```
      subroutine constr( cell, na, isa, amass, xa, stress, fa )
c real*8  cell(3,3)    : input lattice vectors (Bohr)
c integer na           : input number of atoms
c integer isa(na)      : input species indexes
c real*8  amass(na)    : input atomic masses
c real*8  xa(3,na)     : input atomic Cartesian coordinates (Bohr)
c real*8  stress( 3,3) : input/output stress tensor (Ry/Bohr**3)
c real*8  fa(3,na)     : input/output atomic forces (Ry/Bohr)
c integer ntcon        : output total number of position constraints
c                        imposed in this routine
      integer na, isa(na), ntcon
      double precision amass(na), cell(3,3), fa(3,na),
     .                 stress(3,3), xa(3,na), fz
      fz = fa(3,1) + fa(3,2)
      fa(3,1) = fz * amass(1)/(amass(1)+amass(2))
      fa(3,2) = fz * amass(2)/(amass(1)+amass(2))
      ntcon=1
      end
```

NOTE that the input of the routine **constr** has changed with respect to SIESTA versions prior to 1.3. Now, it includes the argument *ntcon*, where the routine should store the number of position constraints imposed in it, as an output. The user should update older **constr** routines accordingly. In the example above, the number of constraints is one, since only the relative z position of two atoms is constrained to be constant.

Example: consider a diatomic molecule (atoms 1 and 2) above a surface, represented by a slab of 5 atomic layers, with 10 atoms per layer. To fix the cell height, the slab's bottom layer (last 10 atoms), the molecule's interatomic distance, its height above the surface and the relative height of the two atoms (but not its azimuthal orientation and lateral position):

```
%block GeometryConstraints
   cellside   c
   cellangle  alpha  beta  gamma
   position  from -1 to -10
   rigid  1  2
   center 1  2   0.0  0.0  1.0
   stress 4 5 6
   routine constr
%endblock GeometryConstraints
```

The first line fixes the height of the unit cell, leaving the width and depth free to change (with the appropriate type of dynamics). The second line fixes all three unit-cell angles. The third line fixes all three coordinates of atoms 1 to 10, counted backwards from the last one (you may also specify a given direction, like in center). The fourth line specifies that atoms 1 and 2 form a rigid unit. The fifth line fixes the center of the molecule (atoms 1 and 2), in the z direction (0.,0.,1.). This vector is given in Cartesian coordinates and, without it, all three coordinates will be fixed (to fix a center, or a position, in the x and y directions, but not in the z direction, two lines are required, one for each direction). The sixth line specifies that the stresses 4, 5 and 6 should be fixed. The convention used for numbering stresses is that 1=xx,2=yy,3=zz, 4=yz,5=xz,6=xy. The list of atoms for a given constraint may contain several atoms (as in lines 4 and 5) *or* a range (as in the third line), but not both. But you may specify many constraints of the same type, and a total of up to 10000 lines in the block. Lines may be up to 130 characters long. Ranges of atoms in a line may contain up to 1000 atoms. All names must be in lower case.

Notice that, if you only fix the position of one atom, the rest of the system will move to reach the same relative position. In order to fix the *relative* atomic position, you may fix the center of the whole system by including a line specifying 'center' without any list or range of atoms (though possibly with a direction).

Constraints are imposed by suppressing the forces in those directions, before applying them to move the atoms. For nonlinear constraints (like 'rigid'), this does not impose the exact conservation of the constrained magnitude, unless the displacement steps are very small.

Default value: No constraints

7.8 Phonon calculations

If **MD.TypeOfRun** = `FC`, SIESTA sets up a special outer geometry loop that displaces individual atoms along the coordinate directions to build the force-constant matrix.

The output (see below) can be analyzed to extract phonon frequencies and vectors with the VIBRA package in the `Util/Vibra` directory. For computing the Born effective charges together with the force constants, see **BornCharge**).

MD.FCDispl (*real length*): Displacement to use for the computation of the force constant matrix for phonon calculations.

Use: Used only if **MD.TypeOfRun** = `FC`.

Default value: `0.04 Bohr`

MD.FCfirst (*integer*): Index of first atom to displace for the computation of the force constant matrix for phonon calculations.

Use: Used only if **MD.TypeOfRun** = `FC`.

Default value: `1`

MD.FClast (*integer*): Index of last atom to displace for the computation of the force constant matrix for phonon calculations.

Use: Used only if **MD.TypeOfRun** = `FC`.

The force-constants matrix is written in file *SystemLabel*.`FC`. The format is the following: for the displacement of each atom in each direction, the forces on each of the other atoms is writen (divided by the value of the displacement), in units of eV/Å^2. Each line has the forces in the x, y and z direction for one of the atoms.

(If constraints are used, the file *SystemLabel*.`FCC` is also written.)

8 TRANSIESTA

The present SIESTA release includes the possibility of performing calculations of electronic transport properties using the TRANSIESTA method. This Section describes how to compile the code to be able to use these capabilities, and a reference guide to the relevant FDF options. We describe here only the additional options available for TranSiesta calculations, while the rest of the Siesta functionalities and variables are described in the previous sections of this User's Guide.

8.1 Brief description

The TRANSIESTA method is a procedure to solve the electronic structure of an open system formed by a finite structure sandwiched between two semi-infinite metallic leads. A finite bias can be applied between both leads, to drive a finite current. The method is described in detail in Phys. Rev. B **65**, 165401 (2002). In practical terms, calculations using TRANSIESTA involve the

solution of the electronic density from the DFT Hamiltonian using Green's functions techniques, instead of the usual diagonalization procedure. Therefore, TranSiesta calculations involve a Siesta run, in which a set of routines are invoked to solve the Green's functions and the charge density for the open system. These routines are packed in a set of modules, and we will refer to it as the 'TranSIESTA module' in what follows.

TranSiesta was originally developed by Mads Brandbyge, José-Luis Mozos, Pablo Ordejón, Jeremy Taylor and Kurt Stokbro (see references). It consisted, mainly, in setting up an interface between Siesta and the (tight-binding) transport codes developed by M. Brandbyge and K. Stokbro. Initially everything was written in Fortran-77. As Siesta started to be translated to Fortran-90, so were the TranSiesta parts of the code. This was accomplished by José-Luis Mozos, who also worked on the parallelization of TranSiesta. The present distribution has been adapted to the new Siesta code structure. With respect to the previous implementations, it has the additional feature of allowing for the use of a k-point sampling other than the gamma point (for the 2D Brillouin zone perpendicular to the transport direction). These modifications, among others, were done by Frederico D. Novaes.

TranSiesta has been recently (Fall 2012) restructured to be more readable from a programmer's point of view. This will result in less maintenance of the code. Several improvements have also been accomplished:

- full parallelization of the Green's function files (GF-files) generation,
- full Green's function calculation at initialization,
- re-implementation of repetition of the electrodes,
- more SCF information about charge redistribution,
- separate k-point file (TSKP instead of overwriting KP),
- changed the `CONTOUR` file to `TSCC`,
- changed the GF files to be named according to the *SystemLabel*,
- general optimizations in many routines.

The above implementations were done by Nick P. Andersen.

8.2 Source code structure

In this implementation, the original TranSiesta routines have been grouped in a set of modules whose file names begin with `m_ts` (such as in e.g. `m_ts_electrode.F90`). Several new subroutines have been added. These modules are located in the `Src` directory. The inclusion of TranSiesta has also required the modification of some of the Siesta routines. Presently, these modifications are controlled by pre-processor compilation directives (such as in `#ifdef TRANSIESTA`). See the next section for compilation instructions.

8.3 Compilation

The standard SIESTA executable (obtained as described in Section 2) does not include the TRANSIESTA modules. In order to use the TRANSIESTA capabilities, you must compile the SIESTA package as indicated in this Section. In this way, the compilation is done using the appropriate preprocessor flags needed to include the TRANSIESTA modules in the binary file. To generate a binary of SIESTA which includes the TRANSIESTA capabilities, just type:

```
$ make transiesta
```

using the appropriate arch.make file for your system (note that you do not need to make any modification on your arch.make file: you can use the same one that you have used to make a standard SIESTA compilation in your system). The Makefile takes care of defining the appropriate preprocessor flag `-DTRANSIESTA` so that the TRANSIESTA modules and modifications are compiled and incorporated into the binary. Upon successful compilation, the binary file `transiesta` will be generated, containing an executable version of SIESTA with TRANSIESTA capabilities.

8.4 Running a fast example

Before giving more detailed explanations about TRANSIESTA, let us start with an example to show the basic operations of a transport calculation. Starting from the top SIESTA directory:

```
$ cd Examples/TranSiesta
```

First it is necessary to do the electrode calculation (see below for details),

```
$ cd Elec
$ mkdir OUT_Test
$ cd OUT_Test
$ cp ../* .
$ transiesta < elec.fast.fdf > elec.fast.out
```

Note that apart from the usual files generated by SIESTA, now you will find the `elec.fast.TSHS` file (in general `<SystemLabel>.TSHS`). This file contains the real-space Hamiltonian and Overlap matrices, together with some other information, that will be used, in the case of electrodes, to calculate the surface Green's functions.

Once the electrode file has been generated, we can perform the TRANSIESTA calculation (where the **SolutionMethod** flag is set to `transiesta`).

```
$ cd ../../Scat
$ mkdir OUT_TS_Test
$ cd OUT_TS_Test
$ cp ../* .
$ cp ../../Elec/OUT_Test/elec.fast.TSHS .
$ transiesta < scat.fast.fdf > scat.fast.out
```

Now the two following files should have been generated, `scat.fast.TSHS` and `scat.fast.TSDE`. The first one contains, as previously mentioned, essentially the Hamiltonian and Overlap matrices, and the `.TSDE` file has the TranSiesta density matrix, the equivalent to the `.DM` file of Siesta. The transmission function and the current are calculated using the `tbtrans` postprocessing tool (below).

Other automated TranSiesta-TBTrans examples can be found in `Tests/TranSiesta-TBTrans`.

8.5 Brief explanation

- Transport calculations involve Electrodes (EL) calculations, and then the Scattering Region (SR) calculation. The Electrodes calculations are usual Siesta calculations, but where a file `<SystemLabel>.TSHS` is generated. These files contain the information necessary for the SR calculation. If both electrodes are identical structures (see below) the same `.TSHS` file can be used to describe both. In general, however, both Electrodes can be different and therefore two different `.TSHS` files must be generated. The location of these Electrode files must be specified in the file FDF input file of the SR calculation (they are usually copied to the same directory where the SR calculation is performed).

- For the SR, TranSiesta starts with the usual Siesta procedure, converging a Density Matrix (DM) with the usual Kohn-Sham scheme for periodic systems. It uses this solution as an initial input for the Green's functions self consistent cycle. As it is known, Siesta stores the DM in a file with extension `.DM`. In the case of TranSiesta, this is done in a file named `<SystemLabel>.TSDE`. In a rerun of the same system (meaning the same `<SystemLabel>`), if the code finds a `.TSDE` file in the directory, it will take this DM as the initial input and this is then considered a continuation run. In this case it does not perform an initial Siesta run. It must be clear that when starting a calculation from scratch, in the end one will find both files, `<SystemLabel>.DM` and `<SystemLabel>.TSDE`. The first one stores the Siesta density matrix (periodic boundary conditions in all directions and no voltage), and the latter the TranSiesta solution. It is a good practice to, when increasing the bias, use as an initial DM a `.TSDE` that had been obtained for a lower voltage. It is also usefull to point out here that the `<SystemLabel>.TSDE` file has the same format as the `<SystemLabel>.DM` file (with extra information appendend in the end). Being so, one can for example use DENCHAR to analyse the non equilibrium charge density.

- As in the case of Siesta calculations, what TranSiesta does is to obtain a converged DM, but for open boundary conditions and possibly a finite bias applied between the Electrodes. The corresponding Hamiltonian matrix (once self consistency is achieved) of the SR is also stored in a `<SystemLabel>.TSHS` file. The transport properties are obtained in a post-processing procedure using the `tbtrans` code (located in the `Util/TBTrans` directory). What `tbtrans` does is, using the `.TSHS` file of the SR obtained with TranSiesta, **and** the Electrode's `.TSHS` files, to calculate the transmission spectrum and the electronic current. The `tbtrans` input file is typically the same as the one that was used for TranSiesta, with the additional `tbtrans` options. It is to be noted that the `.TSHS` files contain all the needed structural information (atomic positions, matrix elements, ...), and so this kind of parameters will not be changed by input (fdf) flags once they are read a `.TSHS` file.

- TranSiesta defines the Left Electrode to be the first atoms specified in the SR `.fdf` file, and the Right Electrode to be the last ones. The transport direction has to be considered to be the the third cartesian axis, the `z` axis. The Left Electrode atoms must have smaller `z` components than the Right Electrode atoms. It is also crucial that the atomic positions specified at the left (right) EL calculation must be equivalent to the left (right) electrode part of the SR setup. Here, equivalent means that they can be made equal by a simple translation in space. It is also possible to use buffer atoms. This is mostly useful for simulations with different Electrodes. In this case, TranSiesta will not consider these atoms, and the buffer atoms are considered only for the initial Siesta calculation, to get a better "bulk-like" environment at the electrodes. Lastly the electrodes must not extend into the transport direction. This is checked for by TranSiesta and the execution will stop if the geometry is badly formatted.

- An important parameter is: `TS.ComplexContour.Emin` It specifies the starting energy for the contour integration. It is a good practice, to start with a Siesta calculation for the SR and look at the eigenvalues of the system. The value of `TS.ComplexContour.Emin` must be (considerably) lower than the smallest eigenvalue obtained with Siesta. This ensures that all the states are considered in the contour integration.

- TranSiesta still assumes periodic boundary conditions in the `xy` directions. For TranSiesta, the specified k-point sampling (of this 2-dimensional Brillouin zone) used in a SR calculation must be the same as the one that was used for the electrodes, if they are different the code will stop. In practice this means that the first and the second lines of the **kgrid_Monkhorst_Pack** block must be the same. In the case of **tbtrans**, the k-point sampling has to be specified also using a **kgrid_Monkhorst_Pack** block (or preferentially: **TBT_kgrid_Monkhorst_Pack**), and can differ from the sampling that was used in the TranSiesta calculation. The convergence of the transmission function with respect to the k sampling can be slower than the one for the density matrix. This means that one may have to increase the number of k-points used in **tbtrans**.

8.6 Electrodes

In order to calculate the electronic structure of a system under external bias, TranSiesta attaches the system to semi-infinite electrodes which extend to the left and right of the contact region. Examples of electrodes would include surfaces, nanowires, nanotubes or even atomic chains. The electrode must be oriented along the z-axis and the unit cell along the z-direction must be large enough so that orbitals within the unit cell only interact with a single nearest neighbor cell (the size of the unit cell can thus be derived from the range of support for the orbital basis functions). TranSiesta will warn if this is not enforced. The electrode description is also used in **tbtrans**. The electrodes are generated by a separate transiesta run on a bulk system. The results are saved in a file with extension `.TSHS` which contains a description of the electrode unit cell, the position of the atoms within the unit cell, as well as the Hamiltonian and overlap matrices that describe the electronic structure of the lead. One can generate a variety of electrodes and the typical use of transiesta would involve reusing the same electrode for several setups. At runtime, the transiesta coordinates are checked against the electrode coordinates and the program stops if there is a mismatch to a certain precision (1e-4 Bohr).

When creating the Green's function one needs the transfer matrix of the electrode. This means that one needs to have a supercell in the electrode calculation in the transport direction. The current implementation checks for this as also checks that there are no more than 2 neighbouring supercells in the transport direction. This is so because the surface Green's function relies on a transfer matrix between principal layers and not across several principal layers. For now TranSiesta will continue but print a warning. The warning will look like this:

```
WARNING: Connections across 2 unit cells or more in the transport direction.
WARNING: This is inadvisable.
WARNING: Please increase the electrode size in the transport direction.
WARNING: Will proceed without further notice.
```

This can possibly lead to an erroneous surface Green's function. If this warning is shown try and increase the electrode layers and use the option **TS.NumUsedAtomsLeft/Right** to limit the used number of atoms in the surface Green's function calculation. An often encountered problem is the ABC stacking of gold. Here, it will almost always give this warning. However, one can overcome this by making the electrode to an ABCABC stacking of 6 atoms, and use **TS.NumUsedAtomsLeft/Right** equal to 4, or 3 (this will still have a little error, however, the surface Green's function has been calculated using the full ABCABC stacking which holds more information).

8.6.1 Repetition

TranSiesta is now capable of utilizing the electrode periodicity. In the test directory a small example of how it is used can be found. For this the following options are needed:

TS.ReplicateA1Left (*integer*): Describing the number of replications of the electrode structure in the A1 direction. This, however, requires an equal replication of the k-grid in the electrode in the A1 direction. A small example.

Consider a full TranSiesta structure with metallic stacking:

ABABAB-D-BABAB

, where D is the device region. Each layer is 3x3 atoms. The unit cell is $9. \times 9.$ Angstroms in the layers perpendicular to the transport direction (for the sake of simplicity). Only the Γ-point is used.

The left/right electrodes (ABAB) can then be setup in the following ways:

1. A full ABAB-stacking of 3x3 atoms per layer, and unit cell size of $9. \times 9.$ Angstroms. Only the Γ-point is used, or
2. an ABAB-stacking of 1x3 atoms per layer, and unit cell size of $3. \times 9.$ Angstroms. Here the k-grid needs to be 3 times as large in the repeated direction, thus: $k_{\mathrm{grid}} = \{3, 1, 1\}$, or
3. an ABAB-stacking of 1x1 atoms per layer, and unit cell size of $3. \times 3.$ Angstroms. Here the k-grid needs to be 3 times as large in each of the repeated direction, thus: $k_{\mathrm{grid}} = \{3, 3, 1\}$.

Notice that the second option is not allowed with interchanging the A1 repetition with the A2 direction, this is an implementation decision.

Each of these setups will yield different size GF files. If limited on space and you require many energy-points for the contour integral, consider creating the electrode in the smallest repeated cell and utilize these settings. If you have a very large system, it could be desirable to limit the repetition as the expansion of the Green's function is done in the SCF TranSiesta cycle. However, it is nearly always preferable to utilize the repetition.

Default value: `1`

TS.ReplicateA2Left (*integer*): See **TS.ReplicateA1Left**, however, this is in the A2 direction.

Default value: `1`

TS.ReplicateA1Right (*integer*): See **TS.ReplicateA1Left**, however, this is for the right electrode.

Default value: `1`

TS.ReplicateA2Right (*integer*): See **TS.ReplicateA2Left**, however, this is for the right electrode.

Default value: `1`

8.7 TranSiesta Options

The FDF options shown here are only to be used at the input file for the scattering region. When using `transiesta` for electrode calculations, only the usual Siesta options are relevant.

8.7.1 General options

SolutionMethod (*string*): Must be set to `transiesta` in order to perform a TranSiesta calculation

Default value: `diagon`

TS.SaveHS (*logical*): Save the Hamiltonian in the file with extension .TSHS. Must be `true` when calculating the electrode Hamiltonian (it is by default). The .TSHS file must also be generated in TranSiesta calculations if tbtrans is to be used after the run.

Default value: `true`

TS.Voltage (*physical*): The voltage applied along the z-direction of the unit cell between the two electrodes.

Default value: `0.0 eV`

TS.MixH (*logical*): During the self consisten cycle, usually the density matrix of previous steps are mixed to give the next density matrix. This flag represents the possibility of mixing the Hamiltonian instead. If used, it may result in faster convergence.

Default value: `false`

TS.UpdateDMCROnly (*logical*): During the TranSiesta (Green's functions) self consistent cycle, it updates only the density matrix elements of the contact region, if set to `true`. The electrodes and coupling terms are kept as the ones obtained in a first Siesta run. If set as `false`, the coupling terms are also updated by the Green's functions density matrix. If a larger number of electrode layers (metallic systems) are included in the contact region, the coupling terms may not need to be updated. If set to `false`, however, may result in larger number of iterations to converge.

Default value: `true`

TS.ReUseGF (*logical*): The generated surface Green's functions of the electrodes depend on the atomic structure, but also on the energy points of the contour (that will depend on the voltage). It is possible to use previously generated .GF files, but care must be taken. Per default it will try to re-use the GF files present. A check against the parameters ensures that they are equivalent (however, the atomic coordinates of the electrodes will not be checked). By setting to `false` the Green's function files will be re-created no matter the existence of GF files. This will save computing time.

Default value: `true`

TS.TriDiag (*logical*): If represented in terms of the left electrode, the contact region and the right electrode, the Hamiltonian is tridiagonal (no interactions between the electrodes). To obtain the Green's function used to compute the density matrix, the essential operation is an inversion of a tridiagonal matrix. This matrix can be inverted directly or by using smaller matrices (due to the tridiagonality). If set to `true`, it is done in this way. Different memory uses and times for the inversion operation can be obtained when using one or the other.

Default value: `false`

8.7.2 Electrode description options

TS.HSFileLeft (*string*): Name of the .TSHS file output from the initial electrode run. N.B.: The program will stop if this file is not found.

Default value: `_NONE_`

TS.GFFileLeft (*string*): Name of the .GF file of the left electrode. N.B.: The program will generate a new one if not found.

Default value: `<SystemLabel>.TSGFL`

TS.HSFileRight (*string*): Name of .TSHS file describing right electrode. See TS.HSFileLeft.

Default value: `_NONE_`

TS.GFFileRight (*string*): Name of the .GF file of the left electrode. N.B.: The program will generate a new one if not found.

Default value: `<SystemLabel>.TSGFR`

TS.NumUsedAtomsLeft (*integer*): The number of electrode atoms to include in the left lead (for example it could be 2 if only the Greens function of the first two layers of a fcc(111) surface is needed and in which case you need 3 atoms in the bulk unit cell to represent the A,B,C,A,.. stacking). Must be less than or equal to the number of atoms in the simple unit cell of the left electrode. If it is less than the number of atoms in the simple unit cell, the last **TS.NumUsedAtomsLeft** atoms are taken. If it is less than the number of atoms in the simple unit cell, the atoms in the left electrode must be ordered according to their coordinate along the z-direction, from smallest to largest.

Default value: `Number of atoms in the simple unit cell of the Left electrode`

TS.NumUsedAtomsRight (*integer*): The number of electrode atoms to include in the right lead. Must be less than or equal to the number of atoms in the simple unit cell of the right electrode. If it is less than the number of atoms in the simple unit cell, the first **TS.NumUsedAtomsRight** atoms are taken. If it is less than the number of atoms in the simple unit cell, the atoms in the right electrode must be ordered according to their coordinate along the z-direction, from smallest to largest.

Default value: `Number of atoms in the simple unit cell of the Right electrode`

TS.BufferAtomsLeft (*integer*): Number of atoms starting from the first atom to neglect in the TranSiesta run.

Default value: `0`

TS.BufferAtomsRight (*integer*): Number of atoms starting from the last atom to neglect in the TranSiesta run.

Default value: `0`

TS.CalcElectrodeValenceBandBottom (*logical*): If true will calculate the valence band bottom of the electrode. This can be used to check for the contour points.

Default value: `false`

TS.ChargeCorrection (*string*): Allows for doing a charge correction in the SCF TranSiesta cycle. The charges will fluctuate during the SCF cycle and this is available for trying to "fix" the charge.

Available methods:

'none' Will not correct charges

'b' or 'buffer' Will correct missing/excess charges in the buffer regions. All excess charge is removed from the buffer atoms with a scaling factor (**TS.ChargeCorrectionFactor**). If there are no buffer atoms TranSiesta will quit during initialization.

Use with care.

Default value: `'none'`

TS.ChargeCorrectionFactor (*value*): The factor for correcting the missing charges. Must be in the range $[0.0; 1.0]$. This option only has meaning if **TS.ChargeCorrection** is not 'none'.

Example: After an SCF cycle, TranSiesta is missing 0.4 electrons. The correction performed will add $0.4 \times$ **TS.ChargeCorrectionFactor** to the system for compensation. This factor can be used to control the fluctuations.

Default value: 0.75

8.7.3 Complex contour integration options

TS.ComplexContour.Emin (*physical*): The starting point of the complex energy contour. In a TranSiesta run this value should be below the lowest energy in the energy spectrum otherwise some charge will be missing in the integration.

Default value: `-3.0 Ry`

TS.ComplexContour.NumCircle (*integer*): Number of points along the arc part of the contour (starting at **TS.ComplexContour.Emin** and ending at $E_F = 0$).

Default value: `24`

TS.ComplexContour.NumLine (*integer*): Number of points on the line part of the contour.

Default value: `6`

TS.ComplexContour.NumPoles (*integer*): Number of Fermi poles that the complex contour should include.

Default value: `6`

8.7.4 Bias contour integration options

TS.BiasContour.Eta (*physical*): Small finite complex part of the real energy contour.

Default value: 10^{-6} `Ry`

TS.BiasContour.Method (*string*): This describes how the points on the real axis contour are chosen. Options are:

- `Sommerfeld`: equally spaced points with Sommerfeld expansion for including the electron temperature.
- `GaussFermi`: Gaussian quadrature weighted with the Fermi distribution function.

Default value: `GaussFermi`

TS.BiasContour.NumPoints (*integer*): Number of contour points on the close-to-real axis part of the contour in the voltage bias window.

Default value: `5`

8.8 Matching TranSiesta coordinates: basic rules

Having discussed the possible input options of TranSiesta here we just list a set of rules to construct the appropriate coordinates of the scattering region. The unitcell most only have one component for the transport direction vector (z-direction). The order of atoms be such that:

- The first **TS.BufferAtomsLeft** atoms will be considered buffer atoms that will not be used in the TranSiesta calculation, but which are used in the Siesta calculation. This number can, of course, be zero.
- The next **TS.NumUsedAtomsLeft** $\times$ **TS.ReplicateA1Left** $\times$ **TS.ReplicateA2Left** correspond to the left electrode atoms.
- The next atoms correspond to the scattering region.
- The next **TS.NumUsedAtomsRight** $\times$ **TS.ReplicateA1Right** $\times$ **TS.ReplicateA2Right** are the right electrode atoms.
- The next **TS.BufferAtomsRight** correspond to atoms that are neglected in the transiesta part of the calculation, only take part in the first Siesta run (only occurs if it is not a continuation run)

The order shown here must also correspond to increasing values of the `z` coordinates of the atoms, in the sense that the left buffer atoms must all have smaller `z` components than the left electrode atoms, and so on. But, within each *"block"*(buffer atoms, or electrode atoms, etc ...), the coordinates do not have to be ordered in any special way (except when using for the electrodes a number smaller than what was used in the electrode's unit cell).

8.9 Output

TranSiesta generates several output files. The output files are named `<SystemLabel>.ext`, defined using the SystemLabel FDF command, and .ext depends on the type of the output. Below we list the .ext files which are specific to TranSiesta. For a description of the other output files, we refer the user to the Siesta manual.

`<SystemLabel>.DM` : The Siesta density matrix. Siesta initially performs a calculation at zero bias assuming periodic boundary conditions in all directions, and no voltage, which is used as a starting point for the transiesta calculation.

`<SystemLabel>.TSDE` : The TranSiesta density matrix and energy density matrix. During a transiesta run, the .DM values are used for the density matrix in the buffer (if used) and electrode regions. The coupling terms may o may no be updated in a TranSiesta run (see **TS.UpdateDMCROnly**).

`<SystemLabel>.TSHS` : The Hamiltonian corresponding to .TSDE, and other information needed by TranSiesta and `tbtrans`.

`<SystemLabel>.TSKP` : The k-points used in the TranSiesta calculation. See Siesta `.KP` file for formatting information.

`<SystemLabel>.TSCC` : The complex contour integration path used for calculating the non-equilibrium density matrix.

8.10 Utilities for analysis: `tbtrans`.

The `tbtrans` code can be found in the directory `Util/TBTrans`.

A much optimized version of `tbtrans` is found in the directory `Util/TBTtrans_rep` which is recommended to be used. Only in this version can the repetition be used. It has been completely rewritten to be as fast as possible as well as keeping the maintenance to a minimum. This version has been completely rewritten by Nick Papior Andersen (referred to by NPA in the following).

It is used in order to obtain, in a post-processing way, the transport properties after a TranSiesta run. It was developed by M. Brandbyge. It has been made to conform with the modifications introduced in TranSiesta in late October 2012.

In order to run it, it requires the electrode's `.TSHS` files (may be just one file if the left and right electrodes are equal), and the scattering region's `.TSHS` file. These are generated as explained above. The location of these files are specified by the (already discussed) **TS.HSFileLeft**, **TS.HSFileRight** input options, and by:

TS.TBT.HSFile (*string*): Scattering region `.TSHS` file.

Default value: SystemLabel`.TSHS`

respectively for the left and right electrodes and the scattering region `.TSHS` file.

The energy scale in `tbtrans` is shifted so that the Fermi level of the system, if no voltage were applied, is zero. When computing the transmission function of a zero bias calculation, the transmission at the Fermi level is then given by $T(E = 0)$. When there is a finite bias, the Fermi energy of the left electrode is placed at $V/2$, and that of the right electrode at $-V/2$.

The voltage is specified by **TS.Voltage**. The energy window and number of points for the computation of the transmission function is specified by

TBT_kgrid_Monkhorst_Pack (*block*): Block equivalent to **kgrid_Monkhorst_Pack**, however, this is used to have a separate k-sampling in the tbtrans utility. If it does not exist it will use **kgrid_Monkhorst_Pack**.

Notice: **TBT_kgrid_Monkhorst_Pack** can only be used with the version created by NPA.

Default value: **kgrid_Monkhorst_Pack**

TS.TBT.Emin (*physical*): Lowest energy value of the computed transmission function.

Default value: `-2.0 eV`

TS.TBT.Emax (*physical*): Highest energy value of the computed transmission function.

Default value: `2.0 eV`

TS.TBT.NPoints (*integer*): Number of energy points of the transmission function between **TS.TBT.Emin** and **TS.TBT.Emax**.

Default value: `100`

TS.TBT.CalcElectrodeValenceBandBottom (*logical*): If true will calculate the valence band bottom of the electrode.

Notice: Can only be used with the NPA version.

Default value: `true`

TS.TBT.COOP (*logical*): Calculate the COOP on the PDOS region. Creates the files:

- `<SystemLabel>.COOP`
- `<SystemLabel>.COOPL`
- `<SystemLabel>.COOPR`

Notice: Can only be used with the NPA version.

Default value: `false`

TS.TBT.AtomPDOS (*logical*): Calculate the *onsite* projected density of states in the PDOS region. Creates the files:

- `<SystemLabel>.TOTDOS`
- `<SystemLabel>.ORBDOS`

Notice, that this will not sum up to PDOS in `.TRANS`, as this is onsite density of states.

Notice: Can only be used with the NPA version.

Default value: `false`

Note that it is important to specify the voltage, since this information is not stored in the `.TSHS` files. The current will be computed using the resulting transmission function, so be sure to make it suited for the integration in the bias window (the energy window defined by **TS.TBT.Emin** and **TS.TBT.Emax** being bigger than or equal to the applied bias).

The k-point sampling is defined by the **TBT_kgrid_Monkhorst_Pack** block, and if that is non-existing it will use the **kgrid_Monkhorst_Pack** block. The averaged (over k-points) transmission function is printed in the file `<SystemLabel>.AVTRANS`.

An additional options is:

TS.TBT.NEigen (*integer*): Number of eigenvalues of the transmission matrix to be computed. If larger than 0 `.TEIG` and `.AVTEIG` will be created which holds the k-point eigenchannels and the averaged eigenchannels, respectively.

Default value: `0`

To summarize, here we give a list of the parameters read by `tbtrans` from the input file (the fdf flags, NPA indicates the version created by Nick Papior Andersen):

- **ElectronicTemperature**
- **TS.Voltage**
- **TS.TBT.ReUseGF** (NPA)
- **TS.UseBulkInElectrodes** (NPA)
- **TBT_kgrid_Monkhorst_Pack** (block) has precedence of **kgrid_Monkhorst_Pack** (NPA)
- **kgrid_Monkhorst_Pack** (block)
- **TS.HSFileLeft**
- **TS.HSFileRight**
- **TS.TBT.HSFile**
- **TS.TBT.Emin**
- **TS.TBT.Emax**
- **TS.TBT.NPoints**
- **TS.TBT.NEigen**
- **TS.TBT.CalcIEig**
- **TS.TBT.PDOSFrom**
- **TS.TBT.PDOSTo**
- **TS.TBT.COOP** (NPA)
- **TS.TBT.AtomPDOS** (NPA)
- **TS.TBT.CalcElectrodeValenceBandBottom** (NPA)
- **TS.BufferAtomsLeft**, **TS.BufferAtomsRight**
- **TS.NumUsedAtomsLeft**, **TS.NumUsedAtomsRight**
- **TS.ReplicateA1Left**, **TS.ReplicateA2Left** (NPA)
- **TS.ReplicateA1Right**, **TS.ReplicateA2Right** (NPA)
- **SpinPolarized**

8.10.1 Output

`tbtrans` generates several output files. The output files are named `<SystemLabel>.`*ext*, defined using the SystemLabel FDF command, and *.ext* depends on the type of the output.

`<SystemLabel>.TRANS` : The transmission for each k point, as well as the total density of states in the scattering region and the projected density of states for all orbitals in the region specified by **TS.TBT.PDOSFrom** and **TS.TBT.PDOSTo**.

`<SystemLabel>.AVTRANS` : The k point averaged transmission and density of states according to `.TRANS`.

`<SystemLabel>.LDOS` : The k point averaged and energy weighted density of states of the left electrode.

`<SystemLabel>.RDOS` : The k point averaged and energy weighted density of states of the right electrode.

8.10.2 Compiling TBTtrans

In the `Util/TBTrans` or `Util/TBTrans_rep` (for the NPA version) directory, simply type `make` if your main SIESTA compilation directory is the top `Obj` directory. If you have used another object directory `MyObjDir`, type `make OBJDIR=MyObjDir`.

9 ANALYSIS TOOLS

There are a number of analysis tools and programs in the `Util` directory. Some of them have been directly or indirectly mentioned in this manual. Their documentation is the appropriate sub-directory of `Util`. See `Util/README`.

10 SCRIPTING

In the `Util/Scripting` directory we provide an experimental python scripting framework built on top of the 'Atomic Simulation Environment' (see `https://wiki.fysik.dtu.dk/ase2`) by the Campos group at DTU, Denmark.

(NOTE: "ASE version 2", not the new version 3, is needed)

There are objects implementing the "Siesta as server/subroutine" feature, and also hooks for file-oriented-communication usage. This interface is different from the SIESTA-specific functionality already contained in the ASE framework.

Users can create their own scripts to customize the "outer geometry loop" in Siesta, or to perform various repetitive calculations in compact form.

Note that the interfaces in this framework are still evolving and are subject to change.

Suggestions for improvements can be sent to Alberto Garcia (albertog@icmab.es)

11 PROBLEM HANDLING

11.1 Error and warning messages

`chkdim: ERROR: In` *routine* `dimension` *parameter* `=` *value*`. It must be ...` And other similar messages.

Description: Some array dimensions which change infrequently, and do not lead to much memory use, are fixed to oversized values. This message means that one of this parameters is too small and neads to be increased. However, if this occurs and your system is not very large, or unusual in some sense, you should suspect first of a mistake in the data file (incorrect atomic positions or cell dimensions, too large cutoff radii, etc).

Fix: Check again the data file. Look for previous warnings or suspicious values in the output. If you find nothing unusual, edit the specified routine and change the corresponding parameter.

11.2 Known but unsolved problems and bugs

- Input (fdf) files with CRLF line endings (the DOS standard) are not correctly read by SIESTA on Unix machines.

  Solution: Please convert to the normal LF-terminated form. This is easy, running for example: `$ dos2unix` *yourinput*`.fdf`

- k-points are not properly generated (`kgrid`) if using a **SuperCell** block with a non-diagonal matrix.

  Solution: Make an empty run with **SuperCell** first to generate the whole geometry, and then run for the large unit cell (without the **SuperCell**) with k-points at will.

- For some systems the program stops with the error message

  `"Failure to converge standard eigenproblem Stopping Program from Node: 0`

  It is related to the use of the Divide & Conquer algorithm for diagonalisation.

  Solution: If it happens, disable Diag.DivideAndConquer and run again.

12 REPORTING BUGS

Your assistance is essential to help improve the program. If you find any problem, or would like to offer a suggestion for improvement, please follow the instructions in the file `Docs/REPORTING_BUGS`

13 ACKNOWLEDGMENTS

We want to acknowledge the use of a small number of routines, written by other authors, in developing the siesta code. In most cases, these routines were acquired by now-forgotten routes, and the reported authorships are based on their headings. If you detect any incorrect or incomplete attribution, or suspect that other routines may be due to different authors, please let us know.

- The main nonpublic contribution, that we thank thoroughly, are modified versions of a number of routines, originally written by **A. R. Williams** around 1985, for the solution of the radial Schrödinger and Poisson equations in the APW code of Soler and Williams (PRB **42**, 9728 (1990)). Within SIESTA, they are kept in files arw.f and periodic_table.f, and they are used for the generation of the basis orbitals and the screened pseudopotentials.
- Routine pulayx, used for the SCF mixing, was originally written by **In-Ho Lee** in 1997.
- The exchange-correlation routines contained in file xc.f were written by J.M.Soler in 1996 and 1997, in collaboration with **C. Balbás** and **J. L. Martins**. Routine pzxc (in the same file), which implements the Perdew-Zunger LDA parametrization of xc, is based on routine velect, written by **S. Froyen**.
- Some standard linear-algebra routines from the **EISPACK**, **BLAS**, and **LAPACK** packages are in several files in Libs and Util/Vibra.
- The serial version of the multivariate fast fourier transform used to solve Poisson's equation was written by **Clive Temperton**.
- Subroutine iomd.f for writing MD history in files was originally written by **J. Kohanoff**.

We want to thank very specially **O. F. Sankey, D. J. Niklewski** and **D. A. Drabold** for making the FIREBALL code available to P. Ordejón. Although we no longer use the routines in that code, it was essential in the initial development of the SIESTA project, which still uses many of the algorithms developed by them.

We thank **V. Heine** for his supporting and encouraging us in this project.

The SIESTA project is supported by the Spanish DGES through several contracts. We also acknowledge past support by the Fundación Ramón Areces.

14 APPENDIX: Physical unit names recognized by FDF

Magnitude	Unit name	MKS value
mass	Kg	1.E0
mass	g	1.E-3
mass	amu	1.66054E-27
length	m	1.E0
length	cm	1.E-2
length	nm	1.E-9
length	Ang	1.E-10
length	Bohr	0.529177E-10
time	s	1.E0
time	fs	1.E-15
time	ps	1.E-12
time	ns	1.E-9
time	mins	60.E0
time	hours	3.6E3
time	days	8.64E4
energy	J	1.E0
energy	erg	1.E-7
energy	eV	1.60219E-19
energy	meV	1.60219E-22
energy	Ry	2.17991E-18
energy	mRy	2.17991E-21
energy	Hartree	4.35982E-18
energy	Ha	4.35982E-18
energy	K	1.38066E-23
energy	kcal/mol	6.94780E-21
energy	mHartree	4.35982E-21
energy	mHa	4.35982E-21
energy	kJ/mol	1.6606E-21
energy	Hz	6.6262E-34
energy	THz	6.6262E-22
energy	cm-1	1.986E-23
energy	cm**-1	1.986E-23
energy	cmˆ -1	1.986E-23
force	N	1.E0
force	eV/Ang	1.60219E-9
force	Ry/Bohr	4.11943E-8

Magnitude	Unit name	MKS value
pressure	Pa	1.E0
pressure	MPa	1.E6
pressure	GPa	1.E9
pressure	atm	1.01325E5
pressure	bar	1.E5
pressure	Kbar	1.E8
pressure	Mbar	1.E11
pressure	Ry/Bohr**3	1.47108E13
pressure	eV/Ang**3	1.60219E11
charge	C	1.E0
charge	e	1.602177E-19
dipole	C*m	1.E0
dipole	D	3.33564E-30
dipole	debye	3.33564E-30
dipole	e*Bohr	8.47835E-30
dipole	e*Ang	1.602177E-29
MomInert	Kg*m**2	1.E0
MomInert	Ry*fs**2	2.17991E-48
Efield	V/m	1.E0
Efield	V/nm	1.E9
Efield	V/Ang	1.E10
Efield	V/Bohr	1.8897268E10
Efield	Ry/Bohr/e	2.5711273E11
Efield	Har/Bohr/e	5.1422546E11
Efield	Ha/Bohr/e	5.1422546E11
angle	deg	1.d0
angle	rad	5.72957795E1
torque	eV/deg	1.E0
torque	eV/rad	1.745533E-2
torque	Ry/deg	13.6058E0
torque	Ry/rad	0.237466E0
torque	meV/deg	1.E-3
torque	meV/rad	1.745533E-5
torque	mRy/deg	13.6058E-3
torque	mRy/rad	0.237466E-3

15 APPENDIX: NetCDF

¿From the NetCDF User's Guide:

> The purpose of the Network Common Data Form (netCDF) interface is to allow you to create, access, and share array-oriented data in a form that is self-describing and portable. "Self-describing" means that a dataset includes information defining the data it contains. "Portable" means that the data in a dataset is represented in a form that can be accessed by computers with different ways of storing integers, characters, and floating-point numbers. Using the netCDF interface for creating new datasets makes the data portable. Using the netCDF interface in software for data access, management, analysis, and display can make the software more generally useful.
>
> [...]
>
> NetCDF is an abstraction that supports a view of data as a collection of self-describing, portable objects that can be accessed through a simple interface. Array values may be accessed directly, without knowing details of how the data are stored. Auxiliary information about the data, such as what units are used, may be stored with the data. Generic utilities and application programs can access netCDF datasets and transform, combine, analyze, or display specified fields of the data. The development of such applications may lead to improved accessibility of data and improved reusability of software for array-oriented data management, analysis, and display.

In the context of electronic structure calculations, such an interface is useful to share pseudopotential, wavefunction, and other files among different computers, regardless of their native floating point format or their endian-ness. At present, some degree of transportability can be achieved by using ascii-binary converters. However, the other major advantage of the NetCDF format, the self-description of the data and the ease of accessibility is of great interest also.

> A netCDF dataset contains dimensions, variables, and attributes, which all have both a name and an ID number by which they are identified. These components can be used together to capture the meaning of data and relations among data fields in an array-oriented dataset. The netCDF library allows simultaneous access to multiple netCDF datasets which are identified by dataset ID numbers, in addition to ordinary file names.

To be able to generate NetCDF files in Siesta, the public domain NetCDF library (V. 3.6.12 or higher recommended) must be installed. It can be downloaded from

`http://www.unidata.ucar.edu/software/netcdf/`.

In the `arch.make` file, the following information must exist:

```
NETCDF_LIBS=-L/path/to/netcdf/library/directory -lnetcdf
NETCDF_INCLUDE=-I/path/to/netcdf/include/directory
DEFS_CDF=-DCDF
```

`$(NETCDF_LIBS)` must be added to the `LIBS` list and `$(NETCDF_INCLUDE)` must be added to the `INCFLAGS` list (or `INCFLAGS` may be set directly). See examples in the Src/Sys directory, and `Src/Sys/DOCUMENTED-TEMPLATE.make`.

In NetCDF versions above 4 the FORTRAN library has been separated from the C. Thus you need

```
NETCDF_LIBS=-L/path/to/netcdf/library/directory -lnetcdf -lnetcdff
```

instead.

(SIESTA used to include an old f90 interface to NetCDF in the Src/NetCDF directory. Current versions of NetCDF now come with their own, so that directory has dissappeared.

While it might seem a hassle to install the library, the added functionality is significant: speedup in diagonalization with k-points by storing the eigenvectors, optional restarts with charge density information instead of a density-matrix, new analysis tools, etc.

16 APPENDIX: Parallel SIESTA

At present, SIESTA has been parallelised with moderate system sizes in mind and is suitable for comensurately moderate parallel computing systems of the type most widely available. A version suitable for massively parallel systems in order to tackle grand challenge problems will hopefully be available in the future.

Apart from the possibility of faster real time performance, there is another major driving force for the use of the parallel version. All significant parts of the code have been written using a distributed data strategy over the Nodes. This means that the use of a parallel machine can allow access to a larger amount of physical memory.

Given the targets for the present version, the strategy for parallelism does not employ spatial decomposition since this is only beneficial for very large problem sizes. Hence the work is divided in 2 ways depending on the section of the code :

- For operations that are orbital based, a 1-D block cyclic distribution has been used to divide the work over processors. This is controlled by the parameter **BlockSize**. For optimal performance, this parameter should be adjusted according to the size of problem and the machine being used. Very small and very large values tend to be inefficient and typically values in the range 8 - 32 tend to be optimal. Parts of the code that parallelise in this way are: evaluation of the kinetic energy, the non-local pseudopotential contribution, determination of the overlap integrals and matrix diagonalisation/order N. Note that for matrix diagonalisation, the default option is now to transform the Hamiltonian and Overlap matrices into a 2-D blocked distribution since this gives better scaling within Scalapack. The 1-D block cyclic data distribution can be maintained by setting the option **Diag.Use2D** to false.

- For operations that are grid based, a 2-D block cyclic distribution over mesh points has been used to divide the work. The mesh is divided in the Y and Z directions, but not currently in the X direction. How the mesh points are divided is controlled by the **ProcessorY** option which must be a factor of the total number of processors. Performance will be optimal when the load is balanced evenly over all processors. For dense bulk materials this is straightforward to achieve. For surfaces, where there is a region of vacuum, it is worth ensuring that the mesh is divided so as to ensure that some processors do not have just vacuum regions. Parts of the code that parallelise in this way are anything connected to the mesh (i.e. within DHSCF), including the evaluation of the Hartree and exchange-correlation energies.

There is also a second mode in which the parallel version can be used. For systems where the number of K points is very large and the size of the Hamiltonian/Overlap matrices is small, then the work can be parallelised over K points. This is far more efficient in the diagonalisation step since this phase becomes embarrassingly parallel once the matrices have been distributed to each Node. This mode is selected using the **ParallelOverK** option.

The order-N facility of SIESTA was rewritten as of version 2.0 to parallelise over spatial regions with a domain decomposition. In the ideal situation, each domain interacts with only the neighbouring domains if the size of the domains is greater than the Wannier function radius

and the range of matrix elements in the Hamiltonian. In order to achieve load balance, it may be advantageous to use smaller domain sizes. The domain size can be controlled through the **RcSpatial** option.

(A new option for domain decomposition, based on an abstract partitioning of the interaction graph associated to the Hamiltonian, has been added as of version 3.1.)

In the current implementation of the domain decomposition parallelisation, both the local elements of the orbital coefficients in the Wannier functions and those connected via the transpose are locally stored on each node in order to minimise communication. However, this leads to greater demands on the memory and works best when the system size to processor ratio is high. Work is in progress to offer a modified algorithm with higher communication, but lower memory demands.

In order to use the parallel version of the code you must have the following libraries installed on your computer :

```
(a) MPI :       The Message Passing Interface library - this allows the
                processors to communicate. Most machine vendors have their
                own implementations available for their own platforms.
                However, there are two freely available versions that can
                be installed :

                MPICH :

                  http://www-unix.mcs.anl.gov/mpi/mpich/

                LAMMPI :

                  http://www.lam-mpi.org/

(b) Blacs :     This is a communications library that runs on top of MPI. Again
                if can be obtained for free from :

                  http://www.netlib.org/

                Both source code and pre-compiled binaries are available.

(c) Scalapack : This is a parallel library for dense linear algebra, equivalent
                to "lapack" but for parallel systems. Once again this is freely
                available as source code or in precompiled form from :

                  http://www.netlib.org/
```

Parallel versions of the files for `arch.make` suitable for a number of systems are provided in the `Src/Sys` directory. Should there be no suitable file there for your system, then the following

are the key variables to be set in the `arch.make` file (see also the commented `arch.make` file in `Src/Sys/DOCUMENTED-TEMPLATE.make`):

```
MPI_INTERFACE=libmpi_f90.a
MPI_INCLUDE=/usr/local/include
DEFS_MPI=-DMPI
#
LIBS= -lscalapack -lblacs -lmpi
```

Here `MPI_INTERFACE` indicates that the interface to MPI provided should be used which handles the issue of the variable type being passed. This will be needed in nearly all cases. `MPI_INCLUDE` indicates the directory where the header file "mpif.h" can be found on the present machine. The environment variable `DEFS_MPI` should always be set to `"-DMPI"`, since this causes the preprocessor to include the parallel code in the source. Finally `LIBS` must now include all the libraries required - namely Scalapack, Blacs and MPI, in addition to any machine optimised Blas, etc.

To execute the parallel version, on most machine, the command will now be of the form :

```
mpirun -np <nproc> siesta < input.fdf > output
```

Where `<nproc>` is the desired number of processors, `input.fdf` is the SIESTA input file and `output` is the name of the output file.

Finally, a word concerning performance of parallel execution. This is a very variable quantity and depends on the exact system you are using since it will vary according to the latency and bandwidth of the communication mechanism. This is a function of the means by which the processors are physically connected and by software factors relating to the implementation of MPI. The one almost universal truth is that, for significant system sizes, parallel diagonalisation becomes the bottleneck and the place where efficiency is most readily lost. This is basically just the nature of diagonalisation, but it is always worth tuning the BlockSize parameter.

17 APPENDIX: File Formats

The file formats in Siesta are in a state of flux. On the one hand, some of the legacy formats were inefficient or incomplete, so new ones have been devised, and appropriate translators provided when necessary. Examples are the WFS (WFSX) wavefunction file, and the HS (HSX) Hamiltonian-and-overlap file. On the other hand, we are introducing new files in netCDF format which facilitate the exchange of information among different computers and offer new functionality in Siesta. Examples are the DM.nc and the .Grid.nc files.

Here we provide an overview of some of the most important files and their associated tools.

- Orbital indexing

  ORB_INDX

  Provides information on the basis orbitals, in the order used in files DM, HS, etc (and internally by Siesta). It is a plain-text file, with information on the meaning of each column at the end of the file.

- Density matrix

  DM

  Traditional Density Matrix (DM) file. The DM is written by default at the end of each SCF step, *after mixing*, so it can be directly used in a restart. It is a binary file.

  DM.nc

  DM in netCDF format. It contains the same information as DM, plus the number of orbitals in the auxiliary supercell and the array `indxuo` which maps the orbitals in the supercell to the unit cell. If the history option is used, this file will store all the DMs in an SCF cycle.

  DMHS.nc

  DMin, DMout, Hamiltonian, and Overlap matrix in netCDF format. If the appropriate fdf history option is used, this file will store the corresponding information for all the steps in an SCF cycle.

  Associated tools:

```
Util/DensityMatrix:  cdf2dm, dm2cdf: DM <--> DM.nc conversion
                     experimental octave/matlab scripts to process DM .nc files
Util/SCF:            experimental python scripts to process DM .nc files
```

- Hamiltonian and overlap matrices

  HS

  The old format, very inefficient in terms of space used. It contains (if produced by a calculation using k-points) also the x_ij array and information about the atomic species and orbitals. As of `siesta-3.0-rc2`, it has been superseded by the HSX format. If some legacy utility needs the HS format, it can be re-generated using the tools in `Util/HSX`.

HSX

The new format, with better packing of binary records.

DMHS

See above

- Wavefunctions

  WFS

  Old format, now superseded by WFSX.

  WFSX

  New format, without redundant information, and in single precision.

  WFS.nc

  A netCDF file to store the eigenvectors (in routine diagk_file) as they are computed, thus saving a second round of diagonalization after the calculation of the Fermi level.

  Associated tools:

```
Util/WFS: readwfx, wfsnc2wfsx, readwf,  wfs2wfsx, wfsx2wfs
```

- Grid magnitudes

  RHO, VT, DRHO, VH...

  These are binary files, read directly by (for example) Andrei Postnikov's utilities.

  ...Grid.nc

  netCDF files which can be directly processed by a number of programs and scripts. The Rho.Grid.nc and DeltaRho.Grid.nc files can also be read by Siesta to start a new SCF cycle.

  Associated tools:

```
Util/Grid:

 Operating on old-style files: grid2val, grid2cube, grid2cdf
 Operating on netCDF files:    average_x.m cdf2xsf,
                               average_z.m  cdf_diff, cdf2grid, cdf_fft

Util/Contrib/APostnikov:       rho2xsf, etc.
```

18 APPENDIX: XML Output

From version 2.0, SIESTA includes an option to write its output to an XML file. The XML it produces is in accordance with the CMLComp subset of version 2.2 of the Chemical Markup Language. Further information and resources can be found at `http://cmlcomp.org/` and tools for working with the XML file can be found in the `Util/CMLComp` directory.

As of June 2009, the engine for CML output production is a subset of the FoX library (see `http://www.uszla.me.uk/FoX`).

The main motivation for standarised XML (CML) output is as a step towards standarising formats for uses like the following.

- To have SIESTA communicating with other software, either for postprocessing or as part of a larger workflow scheme. In such a scenario, the XML output of one SIESTA simulation may be easily parsed in order to direct further simulations. Detailed discussion of this is out of the scope of this manual.
- To generate webpages showing SIESTA output in a more accessible, graphically rich, fashion. This section will explain how to do this.

18.1 Controlling XML output

XML.Write (*logical*): Determine if the main XML file should be created for this run.

Default value: `true`

XML.AbortOnErrors (*logical*): This controls the behaviour of the XML output library when it detects internal errors or erroneous use of the API and is intended to aid debugging. When this option is false, the library will emit a diagnostic message to standard error before stopping execution with a Fortran stop statement. When this option is true, the message is generated but execution will be terminated by generating a runtime exception leading to an abort signal. Depending on the execution environment and compiler options, this can lead to the generation of a core file or stack trace.

Default value: `false`

XML.AbortOnWarnings (*logical*): This controls the behaviour of the XML output library when it detects minor errors and is intended to aid debugging. When this option is false, the library will emit a diagnostic message to standard error before continuing execution. When this option is true, the message is generated but execution will also be terminated by generating a runtime exception leading to an abort signal. Depending on the execution environment and compiler options, this can lead to the generation of a core file or stack trace.

Default value: `false`

18.2 Converting XML to XHTML

The translation of the SIESTA XML output to a HTML-based webpage is done using XSLT technology. The stylesheets conform to XSLT-1.0 plus EXSLT extensions; an xslt processor capable of dealing with this is necessary. However, in order to make the system easy to use, a script called ccViz is provided in `Util/CMLComp` that works on most Unix or Mac OS X systems. It is run like so:

```
./ccViz SystemLabel.xml
```

A new file will be produced. Point your web-browser at `SystemLabel.xhtml` to view the output.

The generated webpages include support for viewing three-dimensional interactive images of the system. If you want to do this, you will either need jMol (`http://jmol.sourceforge.net`) installed or access to the internet. As this is a Java applet, you will also need a working Java Runtime Environment and browser plugin - installation instructions for these are outside the scope of this manual, though. However, the webpages are still useful and may be viewed without this plugin.

An online version of this tool is avalable from `http://cmlcomp.org/ccViz/`, as are updated versions of the ccViz script.

19 APPENDIX: Selection of precision for storage

Some of the real arrays used in Siesta are by default single-precision, to save memory. This applies to the array that holds the values of the basis orbitals on the real-space grid, to the historical data sets in Broyden mixing, and to the arrays used in the O(N) routines. Note that the grid functions (charge densities, potentials, etc) are now (since mid January 2010) in double precision by default.

The following pre-processing symbols at compile time control the precision selection

- Add `-DGRID_SP` to the `DEFS` variable in `arch.make` to use single-precision for all the grid magnitudes, including the orbitals array and charge densities and potentials. This will cause some numerical differences and will have a negligible effect on memory consumption, since the orbitals array is the main user of memory on the grid, and it is single-precision by default. This setting will recover the default behavior of previous versions of SIESTA.
- Add `-DGRID_DP` to the `DEFS` variable in `arch.make` to use double-precision for all the grid magnitudes, including the orbitals array. This will significantly increase the memory used for large problems, with negligible differences in accuracy.
- Add `-DBROYDEN_DP` to the `DEFS` variable in `arch.make` to use double-precision arrays for the Broyden historical data sets. (Remember that the Broyden mixing for SCF convergence acceleration is an experimental feature.)
- Add `-DON_DP` to the `DEFS` variable in `arch.make` to use double-precision for all the arrays in the O(N) routines.

20 APPENDIX: Data structures and reference counting

To implement some of the new features (e.g. charge mixing), SIESTA uses new flexible data structures. These are defined and handled through a combination and extension of ideas already in the Fortran community:

- Simple templating using the "include file" mechanism, as for example in the FLIBS project led by Arjen Markus (`http://flibs.sourceforge.net`).
- The classic reference-counting mechanism to avoid memory leaks, as implemented in the PyF95++ project (`http://blockit.sourceforge.net`).

Reference counting makes it much simpler to store data in container objects. For example, a circular stack is used in the charge-mixing module. A number of future enhancements depend on this paradigm.

Index